ALMA MATER STUDIORUM — UNIVERSITÀ DI BOLOGNA

DEIS – Department of Electronics, Computer Science and Systems
PhD Course in Electronics, Computer Science and Telecommunications

Cycle XXIV
Scientific-disciplinary Sector: 09/H1 ING-INF/05

METAHEURISTICS FOR SEARCH PROBLEMS IN GENOMICS
– NEW ALGORITHMS AND APPLICATIONS –

Candidate

Dott. STEFANO BENEDETTINI

Coordinator:

Chiar.mo Prof. Ing. LUCA BENINI

Supervisor:

Chiar.mo Prof. Ing. ANDREA ROLI

Tutor:

Chiar.mo Prof. Ing. ANTONIO NATALI

FINAL EXAMINATION YEAR 2012

To my family

Stefano Benedettini, March 5, 2012

Abstract

In this thesis we made the first steps towards the systematic application of a methodology for automatically building formal models of complex biological systems. Such a methodology could be useful also to design artificial systems possessing desirable properties such as robustness and evolvability. The approach we follow in this thesis is to manipulate formal models by means of adaptive search methods called metaheuristics. In the first part of the thesis we develop state-of-the-art hybrid metaheuristic algorithms to tackle two important problems in genomics, namely, the Haplotype Inference by parsimony and the Founder Sequence Reconstruction Problem. We compare our algorithms with other effective techniques in the literature, we show strength and limitations of our approaches to various problem formulations and, finally, we propose further enhancements that could possibly improve the performance of our algorithms and widen their applicability. In the second part, we concentrate on Boolean network (BN) models of gene regulatory networks (GRNs). We detail our automatic design methodology and apply it to four use cases which correspond to different design criteria and address some limitations of GRN modeling by BNs. Finally, we tackle the Density Classification Problem with the aim of showing the learning capabilities of BNs. Experimental evaluation of this methodology shows its efficacy in producing network that meet our design criteria. Our results, coherently to what has been found in other works, also suggest that networks manipulated by a search process exhibit a mixture of characteristics typical of different dynamical regimes.

Acknowledgments

I thank Christian Blum for the helpful collaboration and for kindly providing access to the cluster in Barcelona. I also thank Thomas Stützle for his encouraging advices on the Founder Sequence Reconstruction problem and for letting me get in contact with a stimulating research environment such as IRIDIA.

I finally thank Roberto Serra and Marco Villani for providing guidance and fruitful ideas for the work regarding Boolean networks described in the second part of this thesis.

Contents

List of Figures

List of Tables

Introduction

One of the most important goals of the modern science and engineering research is to develop methodological and analytical tools to synthesise models of biological and artificial biologically-inspired systems. Typically, these systems turn out to be *complex*, i.e., they are composed of many entities whose relations among them are deeply non-linear. From a methodological point of view, a purely reductionist, i.e, *divide et impera*, approach fails to provide an accurate description of them. Although inherently challenging, the design of complex systems is one of the main ambitions in scientific and engineering disciplines. Model synthesis, identification and tuning, reverse engineering of biological and social networks, design of self-organising artificial systems are just some of the areas in which scientists are asked to face this issue. Such systems and models are often designed and tuned by means of automatic procedures, some of which can be ascribed to the class of *search methods* [190].

In this thesis we try address the design problem, typically in a biological context; from a more generic point of view, we are interested in the task of *model instantiation*. Before going on, it is best to define some basic concepts so as to uniform the terminology. What we are going to give in the following are not widespread and agreed definitions of the terms—model and model instance—but, rather, a personal reinterpretation based on works by Kauffman [127, 128]. Specifically, we will find these definitions convenient when we will later introduce the *ensemble approach* [128] to biological complex system modeling. By *model* we mean the set of entities and relationships the scientist puts into play that: *(i)* can be used to describe a class of phenomena or systems[1] and; *(ii)* can be expressed by means of a formal language (of which we give some examples below). For instance, example of models are mathematical ones, such as parametric systems of partial differential equations; statistical models, such as Markov Chains and Bayesian networks; computational models, abstract machines like finite state automata, the Turing machine, Petri nets, but also Cellular Automata; formal calculi, such as π-calculus and λ-calculus; logical models, such as first order and propositional logics; in software engineering many modeling languages have arisen, like the renowned UML, but also Entity-Relationship diagrams or the simpler flowcharts. Other examples of models, especially network models tailored to the description of gene regulatory networks, can be found at the end of the first section in [128]. The representation a model provides is, inevitably, imprecise and ignores details to some degree. Ignoring details is not a drawback *per se*; details can be omitted in order to make the model simpler: Newtonian mechanics, although ignores relativistic

[1] What we call model for some could also go by the name of *abstract model*.

effects, is more than adequate for "everyday" subluminal speeds.

Model instantiation refers to the application of entities, concepts and relationships defined in a model to the description of a *specific* system; we call such description *model instance*. Let us clarify with an example. The Lotka-Volterra equations, parametric differential equations that describe predator-prey dynamics in biological systems, are an example of a model; on the other hand, the description of population dynamics of rabbits and foxes in a natural reserve is an example of model instantiation, because it entails specifying the parameters of the predator-prey model valid for that particular environment. Another example might involve one of the previously cited network models listed a the end of section 1 in [128]. For instance, Kauffman suggests that one of the viable models of cell dynamics could be a network with piecewise linear differential equations (Glass networks [90]). If we wanted to use such model to describe one particular cell type, we would have then to specify a topology and all node activations, which in turn are parametrised equations whose parameters are, likewise, to be determined. This distinction between model and model instance is fundamental because, in this thesis, we are concerned with the problem of designing model instances of particular biological systems. For every problem studied in this thesis we, therefore, commit to a particular model; moreover, we do not discuss the adequacy of such model with respect to its descriptive capabilities because it is out of the scope of this thesis. For instance, in Chapter 9 we do not question whether our model of choice, Boolean networks, is a more adequate model for genetic regulatory networks (a topic which is discussed elsewhere [202, 208, 209]) than, for example, differential equations.

We want to stress that, from a methodological point of view, synthesising an artificial systems or instantiating a model of a (complex) system are quite similar. Although at a first glance they may look different, since the first task involves constructing an artifact from scratch, whilst the second has the purpose of describing something that already exists, the scientist undertaking either problems proceeds in the same manner. In both scenarios, the scientist defines some kind of merit factor, i.e., the distance from a set of desired properties, to evaluate an *artifact*. By artifact we mean a model instance or an artificial system, which can be indeed described by means of an instance of a model representing the artifact or the procedure to build it. In the case of system modeling, the artifact is, for instance, a parametric system of partial differential equations or a computational model (such as Boolean networks, as we will see in this thesis); experimental observations, or some kind of prior knowledge on the modeled system, provide, instead, the desiderata, hence the quality function. For instance, in case of the problems tackled in Chapters 3 and 6, the quality function is given by the genetic model, i.e., a conjecture about the behaviour of genetic structures (genes, genotypes, haplotypes, etc.) expressed in formal terms. In the case of a robotic system, the artifact can be a robot controller but also the shape of the robot itself, if we think back at artificial life [139]; the scientist then specifies, by means of a quality function, the requested observable behaviour of its robots.

The approach we follow in this thesis is to manipulate our artifacts (model instances) by means of adaptive search methods called metaheuristics. Briefly, metaheuristics are approximate[2] optimisation algorithms that perform a search

[2]Approximate means that they do not return a proof of optimality.

in the "solutions space", where a "solution" in this context is an artifact. The solution space has a topology, meaning that there is a notion of neighborhood and a notion of "altitude", i.e., the value of the quality function[3]. As we will see in Chapter 1, one of the main characteristic of metaheuristics is that they are able to adapt the search process by exploiting information gathered during the execution of the search itself. This makes metaheuristics capable of exploiting regularities of the solution space and capable of leading the search into areas containing high quality solutions.

Another characteristic that makes the automatic construction/tuning of these kinds of artifacts hard is that they are complex, like the system they try to describe. This entails that even small modifications to a parameter might change the observable properties of the artifact, hence its quality value, in a way that is difficult to predict (strong non-linear effects). For instance, suppose that a model can be described by a tuple of real parameters; when dealing with complex systems, often tuples close in the parameter space correspond to realisations of models far apart in the behaviour space. We will see an example of such property when we examine the problem of synthesising Boolean networks (BNs) in Chapter 9, in which we lack an heuristic that can tell which are the effective modification to perform on a artifact in order to attain some modeling goal.

As we wrote above, the quality function expresses the "goodness" of an artifact according to some model; this deep connection between quality measure and model has an interesting implication. A model, by definition, is a simplification of a real system and tractable models leave out some details from the description of a phenomenon; this entails that an artifact that maximises the quality function can be, at most, as accurate as the model it instantiates. This has important practical repercussion because optimisation of the quality function is often an intractable problem[4]. This consideration motivates the use of incomplete techniques, and metaheuristics in particular. On the one hand, a proof of optimality is not needed, therefore we can trade completeness for efficiency. Moreover metaheuristics architecture is flexible, can be easily integrated with other search techniques and modifications; this is an advantage, because it makes it possible to accommodate refinements to the model.

As a final note, we want to stress that a methodology to automatically synthesise models of complex systems can be useful for the construction of artificial systems. The most striking characteristic that most biological complex systems posses is that of being robust, with respect to small perturbations, and, at the same time, evolvable. Robustness means insensibility to small random fluctuations in the environment (external noise). Evolvability means being reactive to change, that is, an evolvable system is quick to adapt to different environmental conditions and carry out its goal. These properties, often sought by system designers, are difficult to achieve in general. An approach suggested by this thesis is to employ automatic design methodologies to synthesise artifacts with features typical of complex biological systems. The idea is that, if we are able to build complex artificial systems, robustness and evolvability will come "for free". Of course, one of the questions that this thesis proposes is how to characterise complex systems, or in other words, how to measure complexity for

[3] We suppose that higher quality correspond to greater values

[4] For example, from the worst case computational complexity viewpoint, the problem might be $\mathcal{NP}$-hard [86].

a generic system. If we were able to do so, we could indeed define a quality function to drive our design methodology.

Overview of the thesis

This thesis will mainly deal with modeling issues in genomics. In the first part we address two related problems, namely, the Haplotype Inference problem by Pure Parsimony and the Founder Sequence Reconstruction Problem (FSRP), which originate from common premises. Technical advances in sequencing of genetic material has led to a rapid growth of available DNA sequences and haplotyped sequences. With this large amount of information, researchers are interested determining the genetic causes that affect health, common diseases (heart disease, diabetes, cancer, etc.) and responses to drugs and environmental factors [224]. Two problems can be defined that address different but complimentary aspects of this research. The first one is Haplotype Inference that consists in inferring the genetic variations (i.e., differences in the genome) of diploid organisms (such as humans) responsible for specific characteristics, such as the one listed above. The second one is the FSRP whose objective is to study the evolutionary history of a population of individuals given a set of their haplotyped sequences. The goal of FSRP is to find a set of founder sequences that, through recombinations and mutations, explains the sequences in the current population.

In the second part, we concentrate on a computational model of gene regulatory networks (GRNs). Briefly, a GRN is a collection of DNA segments (genes), interpreted as nodes in the network, whose interactions, mediated by proteins, dictates cellular behaviour. Many models have been proposed to capture the dynamics of GRNs, ranging from differential equations to discrete and continuous networks. One notable model, which is also the one studied in this thesis, is the one of Boolean networks, first proposed by Kauffman [125].

This thesis is structured as follows.

Chapter 1 is a brief introduction to the topic of metaheuristics. It reviews concepts such as combinatorial optimisation problem, local search, neighbourhood definitions etc. Its main purpose is to define common terminology used throughout the thesis. First we present metaheuristics in general terms (Section 1.1), then we discuss trajectory methods (Section 1.2) and population algorithms (Section 1.3), such as Ant Colony Optimisation (ACO) and Genetic Algorithms (GAs). Although it contains basic concepts, we advise the metaheuristic expert to at least skim through this chapter.

In Chapter 2 we first introduce the Haplotype Inference Problem: in Section 2.1 we summarise the biological foundations of the problem and review the literature on the subject; Section 2.2 proposes a mathematical formulation and defines important concepts such as complementarity and compatibility graph, all used and eventually extended in the chapters to come. We finally overview alternative formulations of Haplotype Inference that correspond to different real-world scenarios (Section 2.3).

In Chapter 3 we tackle Haplotype Inference and propose a hybrid metaheuristic which integrate ACO and Tabu Search. In Section 3.1 we describe and discuss Clark's Rule, a well-known greedy heuristic for Haplotype Inference, and, in particular, we point out its drawbacks. The ACO algorithm we propose (Section 3.2.1) takes inspiration from Clark's Rule, while, at the same

time, overcomes its deficiencies. Section 3.2.2 details instead our Tabu Search. The hybrid is extensively tested and analysed on a large number of well-known benchmarks (Section 3.3); in particular, in Section 3.3.2 we show the favorable comparison with state-of-the-art techniques for Haplotype Inference by parsimony. This chapter concludes with an analysis of the instance structure (Section 3.4) in which we strive to determine what are the structural characteristics of an instance that make it difficult for our algorithm to solve.

Chapter 4 introduces Master-Slave Genetic (MSG) framework, a GA variant suited to combinatorial problems for which a parametrized constructive procedure is available. We detail the framework in Section 4.2 and, with the aim of demonstrating its general effectiveness and versatility, we apply it to three hard combinatorial problems, namely, Haplotype Inference (Section 4.3), Capacitated Vehicle Routing (Section A.1) and Capacitated Minimum Spanning Tree (Section A.2).

Chapter 5 discusses alternative approaches to Haplotype Inference and consists mainly of two contributions. The first one regards an alternative model that makes use of constraint programming techniques (Section 5.1); it is shown that such approach can cope with the presence of unknown sites in an instance, as opposed to the difficulties of the other metaheuristics presented in Chapter 3 and 4, based on a simpler model. The second contribution (Section 5.2) is a clique finding algorithm in the compatibility graph of an instance.

Chapter 6 is dedicated to the description of the Founder Sequence Reconstruction Problem and the proposed metaheuristic solver. In this chapter we present a fast Iterated Greedy heuristic and a state-of-the-art hybrid algorithm based on Large Neighbourhood Search (LNS) framework. Both algorithms encapsulate and enhance existing procedures available in the literature, thereby showing the flexibility of metaheuristics. Sections 6.1 and 6.2 open the chapter with biological motivations and a mathematical formulation, respectively. After an overview of the literature (Section 6.3), we start detailing our Iterated Greedy (Sections 6.4 and 6.5) and discuss experimental results of its application (Section 6.6). The chapter concludes with Section 6.7, which explain our LNS metaheuristic, and Sections 6.8 and 6.9 devoted to extensive analysis of LNS results and its comparison with existing techniques.

Chapter 7 is a brief introduction to synchronous deterministic Boolean network models. Much like we do in Chapter 1, in this chapter we establish common terminology and concept used in the remainder of the thesis (Section 7.1). We define the concepts of trajectory, attractor cycle and basin of attraction and discuss the implications of synchronicity and determinism. Section 7.1.1 explores in further detail Random Boolean Networks (RBNs), a well-known specialisation of BNs which has lately garnered attention as model of GRNs. Finally, Section 7.2 introduces and motivates our approach to biological modeling which is rooted in the ensemble approach.

Chapter 8 presents the Boolean Network Toolkit, a Boolean network software simulator, an important contribution developed in the making of the work described in this thesis. The chapter opens by stating requirements of such simulator (Section 8.1) and proceeds with giving an overview of available simulation tools (Section 8.1.1). In Section 8.2 we motivate our design choices and show how such decisions allow us to have a piece of software which satisfies our requirements and is, at the same time, flexible and extensible. Finally, Section 8.3 demonstrates some typical use cases.

Chapter 9 is devoted to the presentation of our automatic design methodology (Section 9.1.2) and its demonstration in four abstract use cases. In Section 9.2 we study the problem of generating BNs with required attractor length by means of a GA. In Section 9.3 we want to generate a Boolean networks such that requirements on its trajectory are satisfied. Section 9.4 presents a study on the adequateness of RBNs as models of cellular dynamics. In particular, attractors in RBNs are the key elements in the models because they correspond to cellular types. Nevertheless, one of the criticism to RBNs as models of biological systems is that real systems are not synchronous; moreover, if we renounce to synchronicity, not only RBNs become more difficult to analyse, but also the cell type/attractor correspondence is not applicable any more: the very same attractors found in synchronous RBNs are no longer distinguishable if a different update scheme is used. From the experiments we discover that RBNs are characterised by a landscape of similar attractors. Such landscape can potentially change drastically under a different update strategy. In Section 9.5 we apply our methodology to synthesise networks with a landscape of attractor as varied as possible, according to some similarity measure, with the aim of closing this gap between synchronous Boolean networks and biologically plausible networks. Finally, Section 9.6 we investigate the possibility of employing BNs as learning systems. Our aim is to train BNs to solve the Density Classification Problem (DCP). Briefly, in this context a BN is a black-box that, we asked a question in the form of a fixed-size binary vector, it replies whether the answer contains more 1s or more 0s: in other terms, a BN is required to *classify* an input. Although the tasks taken into consideration may seem abstract, they are in fact quite challenging and represent a test bed for our methodology.

Contributions

This thesis contains contributions to the field of applied metaheuristics.

- An effective hybrid metaheuristic algorithm for tackling large-scale instances of the Haplotype Inference by pure parsimony is proposed. The algorithm is competitive with the state of the art and, in in some of the adopted benchmarks, it proves superior with respect to solution quality (Chapter 3).

- The MSG algorithmic framework based on GAs is introduced and applications to Haplotype Inference and graph problems are shown. MSG is a general and flexible approach suited to problems for which a parametric constructive heuristic is available, and, in this respect, is similar to Ant Colony Optimisation. MSG has been successfully applied to a variety of algorithms, namely, the Haplotype Inference by parsimony (Chapter 4), the Capacitated Vehicle Routing Problem and the Capacitated Spanning Tree Problem (Chapter A).
  Moreover, `EasyGenetic`, a software library to rapidly develop MSG algorithms, is briefly presented.

- A new formulation of the Haplotype Inference is suggested. This formulation is different than the more familiar one presented in Chapter 2 in

that it employs constraint optimisation techniques, which make it potentially more effective on a variant of the problem that models unknown information (Chapter 5).

- An algorithm for finding cliques in the compatibility graph is proposed (Chapter 5).

- A hybrid state-of-the-art metaheuristic algorithm for the Founder Sequence Reconstruction Problem is introduced. This algorithm is based on the Large Neighbourhood Search framework and is detailed in Chapter 6.

- The Boolean Network Toolkit, an efficient and extensible open-source simulator for BNs, is described and some use cases are presented (Chapter 8).

- Four applications of our methodology to automatically design BNs with desired properties are shown and discussed. In the last case study we are able to synthesise a BN capable of solving the Density Classification Task, a challenging problem for learning systems (Chapter 9).

Some of the topics treated in this thesis already led to papers appeared in conference proceedings or journals and the related chapters have a similar content, with the exception of Chapters 5 and 8 which contain original unpublished research.

- Journal papers:

  - Battarra, M., Benedettini, S., and Roli, A. (2011). Leveraging saving-based algorithms by master-slave genetic algorithms. *Engineering Applications of Artificial Intelligence*, 24:555–566

  - Roli, A., Benedettini, S., Stützle, T., and Blum, C. (2012). Large neighbourhood search algorithms for the founder sequence reconstruction problem. *Computers & Operations Research*, 39(2):213–224

- Conference papers:

  - Benedettini, S., Roli, A., and Gaspero, L. (2008b). Two-level ACO for haplotype inference under pure parsimony. In *Proceedings of the 6th international conference on Ant Colony Optimization and Swarm Intelligence*, ANTS '08, pages 179–190. Springer Berlin / Heidelberg

  - Benedettini, S., Di Gaspero, L., and Roli, A. (2008a). Towards a highly scalable hybrid metaheuristic for haplotype inference under parsimony. *Hybrid Intelligent Systems, International Conference on*, pages 702–707

  - Benedettini, S., Roli, A., and Di Gaspero, L. (2009b). EasyGenetic: A template metaprogramming framework for genetic master-slave algorithms. In Stützle, T., Birattari, M., and Hoos, H., editors, *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics*, volume 5752 of *Lecture Notes in Computer Science*, pages 135–139. Springer Berlin / Heidelberg

  - Benedettini, S., Blum, C., and Roli, A. (2010). A randomized iterated greedy algorithm for the founder sequence reconstruction problem. In Blum, C. and Battiti, R., editors, *Proceedings of the Fourth Learning and Intelligent OptimizatioN Conference – LION 4*, volume 6073 of *Lecture Notes in Computer Science*, pages 37–51. Springer, Heidelberg, Germany
  - Roli, A., Arcaroli, C., Lazzarini, M., and Benedettini, S. (2009a). Boolean networks design by genetic algorithms. In [229], page 13
  - Roli, A., Benedettini, S., Serra, R., and Villani, M. (2011a). Analysis of attractor distances in Random Boolean networks. In Apolloni, B., Bassis, S., Esposito, A., and Morabito, C., editors, *Neural Nets WIRN10 – Proceedings of the 20th Italian Workshop on Neural Nets*, volume 226 of *Frontiers in Artificial Intelligence and Applications*, pages 201–208. IOS Press
  - Benedettini, S., Roli, A., Serra, R., and Villani, M. (2011). Stochastic local search to automatically design Boolean networks with maximally distant attractors. In Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A., Merelo, J., Neri, F., Preuss, M., Richter, H., Togelius, J., and Yannakakis, G., editors, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 22–31. Springer, Heidelberg, Germany

- Technical reports:

  - Benedettini, S., Di Gaspero, L., and Roli, A. (2009a). Genetic master-slave algorithm for haplotype inference by parsimony. Technical Report DEIS-LIA-09-003, University of Bologna (Italy). LIA Series no. 93
  - Benedettini, S., Roli, A., and Di Gaspero, L. (2009c). Easygenetic: A template metaprogramming framework for genetic master-slave algorithms. Technical Report DEIS-LIA-09-005, University of Bologna (Italy). LIA Series no. 95

Part I

Combinatorial biological problems

Chapter 1

Brief Introduction to Metaheuristics

The aim of this chapter is to acquaint the reader with the topic of metaheuristics. Metaheuristics are frameworks for developing adaptive—and usually approximate—search algorithms for tackling hard combinatorial optimisation problems. Examples of metaheuristics are Simulated Annealing, Tabu Search, Iterated Local Search, Ant Colony Optimisation and Genetic Algorithms. Metaheuristics are a vast and thriving research area; as such, a thorough and complete overview of the techniques and analytic tools in the literature would be outside the scope of this thesis. The purpose of this chapter is to introduce basic terminology and definitions useful to comprehend the algorithms described throughout this dissertation. The novice to the field is, thus, encouraged to read this chapter, whilst the expert is advised to skim through it to familiarise with the terminology.

Along with the preliminary definitions in Section 1.1, in this chapter we present some of the metaheuristic algorithms used in this thesis; in Section 1.2, devoted to trajectory methods, we describe Tabu Search, used primarily in Chapter 3; Section 1.3 introduces population methods and two important applications: Ant Colony Optimisation, principally utilised in Section 3, and Genetic Algorithms, employed in Chapters 4 and 9.

A great number of works is available in literature; for a comprehensive survey of metaheuristics techniques we suggest [31], while for an in-depth reading we recommend the interested reader to look into the book by Hoos and Stützle [110].

1.1 Preliminary definitions

Metaheuristics are general search strategies upon which a specific algorithm for solving an optimisation problem can be designed. A combinatorial optimisation problem (COP) can be defined by the following entities:

- a set of decision variables $X = \{x_1, x_2, \ldots, x_n\}$ with domains $\{D_1, D_2, \ldots, D_n\}$;
- a set of constraints on the decision variables Ω;
- an objective function to be minimised $f : D_1 \times D_2 \times \ldots \times D_n \to \mathbb{R}^+$

The set of all feasible assignment is called *search* (or *solution*) *space* and is denoted by $\mathcal{S} = \{s = \langle (x_1, v_1), (x_2, v_2), \ldots, (x_n, v_n) \rangle \mid v_i \in D_i \wedge s \text{ satisfies } \Omega\}$.

Metaheuristics are popular methods to undertake $\mathcal{NP}$-c COPs and in some contexts their usage provides important advantages with respect to exact techniques. The most important one is that, contrarily to what complete techniques do, they *do not* return a proof of optimality with the result. Looking for a (proven) optimum of the objective function can require, in the worst case, exponential time; metaheuristic techniques trade optimality with efficiency and, as such, are preferable to complete ones in instances where an optimality proof is not required. Examples of such cases are the `HIP` in Chapter 3 and the automatic design of Boolean networks in Chapter 9.

Metaheuristic algorithms are useful when one or more of the following conditions apply:

- an optimality proof is not required;
- efficient complete techniques do not exist;
- it is imperative to find a solution within a short deadline;
- the algorithm is required to return a solution regardless the runtime limit (*any-time solver* property).

Structurally, the abstract architecture of metaheuristic algorithms can be summarised as follows. A metaheuristic is composed of two interacting levels: a lower level and an upper level. The lower level consists of a search strategy which is typically application specific and integrates heuristic information about the problem itself. The upper level is responsible of guiding the underlying search process by employing information gathered during the execution of the lower-layer algorithm itself and stored in some kind of memory; we can say that the upper level embodies a sort of problem-agnostic learning mechanism that makes the whole algorithm (lower plus upper levels) adaptive. One of the most important aspects of metaheuristic design is to carefully balance two key conflicting properties: *exploitation*, or *intensification*, and *exploration* [31]. Exploitation means intensifying the search, i.e., spend more time/computational resources, in promising areas of the solution space; exploration means diversifying the solutions explored so that new areas of the search space are probed and the algorithm does not stay confined in small areas of the solution space—or, in an unfortunate case, even get stuck on a single local minimum without escaping.

1.2 Trajectory Methods

Trajectory methods, also known as *local search* or *perturbative search*, are search processes that iteratively modify the current candidate solution, also called *incumbent solution* or only *incumbent*, by applying *move operators* trying to follow trajectories in the search space leading to good solutions. Local search algorithms are usually stochastic as they involve decisions taken according to a probabilistic distribution.

More formally, we can say that a local search, starting from an initial solution $s_0 \in \mathcal{S}$, walks a trajectory in the state space encountering each iteration

$t \in \mathbb{N}$ solutions $s_0, s_1, \ldots, s_t$. The solution returned is the best one in the trajectory (sometimes can be s_t). The choice of the next solution to visit, or *move*, is done by means of a (possibly stochastic) selection procedure of the picks a new solution in the neighbourhood of the current one. Mathematically, a neighbourhood is a function $\mathcal{N} : \mathcal{S} \Rightarrow 2^{\mathcal{S}}$ which takes a solution and returns one of the possible subset of $\mathcal{S}$ (indicated as $2^{\mathcal{S}}$); notice that neighbourhood relation does not need to be symmetric or transitive.

The neighbourhood relation coupled with the objective function imposes a topology on the search space, or, in other words, introduces the concepts of "being close to" and "altitude". More precisely, the search space becomes a *search graph*, that is, an oriented graph where each vertex is a solution and there exist an edge (s_i, s_j) if $s_j \in \mathcal{N}(s_i)$; moreover each solution is labeled with its value, thereby defining intuitive concepts such as "slopes" (edges that lead to solutions with different values),"plateaus " (close solutions with equal value) and "valleys" (local minima, i.e., solutions s^* such that $\forall s \in \mathcal{N}(s^*), s^* < s$).

From a more practical point of view, many successful metaheuristics are based on the iterative perturbation of a current candidate solution; notable instances of this scheme are Simulated Annealing and Tabu Search. The search process starts from an initial candidate solution s_0 and iteratively produces new candidates by (slightly) perturbing the current one, until some termination criterion is met (e.g., the computation time limit is reached). In other words, the neighbourhood structure is implicitly defined by the perturbation. Different search strategies can be defined by instantiating the two basic choices in the scheme, i.e., the generation and the choice of the next possible candidate solution and the acceptance criterion. These search strategies have been extended and improved, for example by adding advanced exploitation and exploration strategies [32].

As an example of these techniques we describe Tabu Search in more detail.

1.2.1 Tabu Search

Tabu Search [91] is a local search algorithm which exhaustively explores the neighborhood of the current solution, by trying all the possible moves, and chooses as new solution the best among the neighbours evaluated. The peculiarity of Tabu search is that the neighbourhood is restricted by forbidding recently performed moves in order to prevent cycling. The basic idea is to utilise a short-term memory in the form of a *tabu list* $\mathcal{L}$ which memorises the last visited solutions: a solution becomes the new incumbent only if is not contained in $\mathcal{L}$. This entails that a solution can be visited again no earlier than a number of iterations equal to the length of $\mathcal{L}$.

The basic Tabu Search method has been extended in numerous ways; for instance, the length of the is made variable in an interval. Another possibility is to memorise solution attributes instead of solutions; for instance, in the case of a Knapsack problem, $\mathcal{L}$ could contain a list of prohibited objects so that solutions that contain such object are forbidden from being visited. An often employed variant is to accept tabu solutions only if they improve on the incumbent, i.e., they are "downhill" moves.

1.3 Population Methods

Population algorithms are a class of metaheuristics that, as opposed to trajectory methods, at every iteration they manipulate a set of solutions. Some prominent examples of population algorithms are Ant Colony Optimisation, Genetic Algorithm. Population methods take inspiration from collective complex biological phenomena: Ant Colony is inspired by the foraging behaviour of worker ants and Genetic Algorithms from the Theory of Evolution It is important to stress that, although these algorithms borrow ideas and also terminology from biological sciences, they are by no means models of biological systems.

From a more formal point of view, these search strategies actually perform a biased sampling of the search space; the parameters of the probabilistic model used for sampling are iteratively adapted in order to concentrate the search in promising areas of the search space. Population methods adhere to the *model based search* [249] framework, which can be summarised by these three entities:

Model: a probability distribution on the space of solutions;

Sampling: a population of samples is generated;

Update: samples evaluation is used to modify the model.

In this respect, population methods are more akin to statistical methods such as machine learning.

1.3.1 Ant Colony Optimisation

Ant Colony Optimisation [65, 220] is an algorithmic framework inspired by the foraging behaviour of worker ants which are able to construct the shortest path from their nest to a food source. Such behaviour is remarkable because the ant system lacks centralised control and its participants are incapable of direct long-range interactions (only local interactions with the environment) and are constrained by a limited local knowledge of the environment.

Interaction is mediated by means of artifacts that actively modify the environment. Ants are capable of depositing trails of *pheromone*, a chemical substance they produce, and are able to sense concentrations of pheromone left by other workers; pheromone is also volatile and, in time, it evaporates until trails fade away. When an ant has to decide which path to follow, it stochastically chooses the path with higher pheromone intensity.

The interplay between ants, pheromone trails and evaporation makes it possible to have a system that "converges" to the shortest path between food and nest. Shorter paths attract more ants which, in turn, deposit more pheromone thereby making the trails stronger; longer paths are characterised by weak pheromone concentrations due to evaporation and because trails are reinforced less often: in the long run (we could say, after a transient), long paths are "forgotten" by the system. In a way, pheromone artifacts are a sort of long-term distributed memory.

This kind of ant system is characterised by two opposed forces: positive and negative feedback. Positive feedback rewards optimal choices (shorter paths have stronger pheromone trails), whilst negative feedback (evaporation) is the mechanism by which adaptation is achieved. For instance, depleted food sources will not be visited anymore because trails leading to them will not be reinforced.

Algorithm 1 ACO high-level framework.

```
1: while termination conditions not met do
2:    A ← generateSolutions()
3:    pheromoneUpdate(A)
4:    globalActions() {optional}
5: end while
```

The evolution of an ant system is typical of a complex system. Initially the system is in equilibrium, but, thanks to random fluctuations[1], equilibrium is broken and the system converges to a stable dynamical state which minimises system energy: the shortest path.

From an algorithmic point of view, ACO is a constructive algorithm which exploits a pheromone model to guide the solution construction; ACO high-level framework is depicted in Algorithm 1.

generateSolutions() repeatedly applies a constructive heuristic to obtain a population of solutions $\mathcal{A}$. The heuristic constructs a solution by making stochastic choices based on pheromone values. From an abstract point of view, a solution is constituted of components c_i, for instances, cities to visit in the Traveling Salesman Problem or objects in the 01-Knapsack Problem. A partial solution s_p is extended by a component c_k to be picked according a probability distribution $\mathbf{p}$ determined by the following formula:[2]

$$\mathbf{p}(c_k|s_p) = \begin{cases} \dfrac{[\eta_k]^\alpha \cdot [\tau_k]^\beta}{\sum\limits_{c_i \in J(s_p)} [\eta_i]^\alpha \cdot [\tau_i]^\beta} & \text{if } c_k \in J(s_p) \\ 0 & \text{otherwise} \end{cases}$$

where we denote with η_i and τ_i respectively the heuristic problem-specific information and the pheromone value associated to component c_i. Exponents α and β are algorithm parameters to be configured which weight the influence of heuristic information and pheromone learning mechanism. With $J(s_p)$ we denote the set of solution components on which $\mathbf{p}$ is defined. Usually $J(s_p)$ takes into account feasibility constraints which are dependent on the current partial solution and, in a way, defines a sort of neighbourhood of s_p; for example, in the TSP a city must not be visited twice.

The other main component is pheromoneUpdate() which models pheromone deposit and evaporation. When all solutions have been constructed, pheromone values are updated according the following formula:

$$\tau_i \leftarrow (1-p) \cdot t_i + \sum_{s \in \mathcal{A}} \Delta\tau_i^s$$

where

$$\Delta\tau_i^s = \begin{cases} F(s) & \text{if } c_i \text{ is a component of } s \\ 0 & \text{otherwise} \end{cases}$$

[1] In every choice an ant makes there is always a small amount of randomness.

[2] This is of course one of the possible instantiation of the fundamental mechanics and one of the most natural straightforward formulations because *(i)* a solution component could be in principle anything and; *(ii)* many variations of the $\mathbf{p}$ formula and the pheromone update strategy exist.

Algorithm 2 Genetic Algorithm high-level framework.

```
1: P ← GenerateInitialPopulation()
2: Evaluate(P)
3: while termination conditions not met do
4:    P′ ← Crossover(P)
5:    P″ ← Mutate(P′)
6:    Evaluate(P″)
7:    P ← Select(P″,P)
8: end while
```

where p is the evaporation parameter, and function F is called quality function ad satisfies $f(s) < f(s\prime) \Rightarrow F(s) \geq F(s\prime), \forall s \neq s\prime$. F is similar to the concept of fitness function in the context of Genetic Algorithms.

Finally, globalActions() encompass centralised actions that go beyond the ant metaphor, such as the execution of a local search to improve a subset of solution or the further increase of pheromone values on those components belonging to the best found solution.

1.3.2 Genetic Algorithms

Genetic algorithms (GAs) belongs to the broad family of evolutionary computation methods [165] and have been successfully applied as search techniques for several decades [92, 108, 157]. Inspired by Darwin's theory of selection and natural evolution, a GA evolves a population of candidate solutions to a problem by iteratively selecting, recombining and modifying them. The driving force of the algorithm is selection, that biases search toward the *fittest* solutions, i.e., those with the highest objective function value.

The general scheme of GAs is illustrated in Algorithm 2. The algorithm iteratively generates a new population of candidate solutions by applying operators such as selection, mutation and recombination to the current population.

The function Evaluate(P) computes the fitness of each individual of population P. The fitness function is positively correlated with the objective function, that quantifies the quality of a candidate solution and it is usually normalised in the range $[0, 1]$.

Besides fitness function, in GAs three main aspects are subject to designer's intervention: solution representation, recombination operators and selection strategy. Typically GAs do not directly manipulate solutions but rather ensembles (populations) of *solutions representations*, usually called *genomes* or *individuals* according to the biological metaphor. Normally, solutions are encoded into linear genomes, i.e., lists of values, but sometimes more complex structures are employed such as bi- or three-dimensional genomes or, in the case of Genetic Programming [134], whole trees. The impact of solution encoding in the search process, which mirrors the relationship between *genotype* and *phenotype* in biology, is subject of study in the evolutionary algorithm community.

Another key role is assumed by recombination operators, i.e., the ways to generate new solutions from the current population. To parallel the genetic metaphor, two kinds of operators are used; the first one, called *crossover*, takes

a number (usually two) genomes (parents) and produces a new individual containing features from both parents. The second one, called *mutation*, applies a stochastic perturbation to a genome; in this respect is closer to the definition of move introduced for local search methods. Recombination operators are repeatedly applied to the current population until an *offspring* population is constructed (Line 5 of Algorithm 2).

The last aspect to examine is the selection strategy (Line 7), i.e., the algorithm that chooses the individual from the current population (P) and the offspring (P'') to be carried over to the next generation. Typically, selection takes into account the fitness values of the genomes; for instance, steady-state selection carries over only the best solutions from the union of P and P''. From a biological standpoint, selection, alongside the fitness function, plays the role of the environment that rewards the better individuals.

1.4 Conclusions and discussion

In this chapter we gave a high-level description of metaheuristic algorithms and we introduced a common vocabulary of terms useful in chapters to come. We examined the two classes of metaheuristic techniques, trajectory methods and population algorithms, we pointed out their peculiarities and provided some examples; specifically, for trajectory methods we described in more detail Tabu Search, while for population algorithms we focused on Ant Colony Optimisation and Genetic Algorithms.

Chapter 2

The Haplotype Inference Problem

In this chapter we present the first of the two biological problems studied in this thesis: the Haplotype Inference Problem. This is an introductory chapter to the problem where we give a brief biological background (Section 2.1) and provide a mathematical formulation (Section 2.2). Section 2.2 is particularly important because it introduces terminology and formal aspects of the problem that will be useful throughout the first part of this thesis, and will be further expanded in Chapter 5. Finally, Section 2.3 summarises relevant alternative formulations of Haplotype Inference that correspond to different real-world scenarios.

2.1 Biological introduction and motivation

The role of genetic variation and inheritance in human diseases is extremely important, though still largely unknown [224]. To this aim, the assessment of a full Haplotype Map of the human genome has become one of the current high priority tasks of human genomics [223]. Diploid organisms, such as mammals, have their genetic material arranged in pairs of chromosomes, one inherited from the father and the other from the mother. A haplotype is one of the two non identical copies of a chromosome of a diploid organism. The fine-grained information conveyed by haplotypes makes it possible to perform association studies for the genetic variants involved in diseases and the individual responses to therapeutic agents. We know that the majority of human genome is shared among all individuals and only a small percent accounts for all variations. Out of this small fraction, the most important and common DNA variations considered by biologists are the *Single Nucleotide Polymorphisms* (SNPs pronounced *snip*), which occur when a nucleotide in the DNA sequence is replaced by another one. In most of the cases, in a SNP site only two nucleotides occur. Within a population, the most frequent nucleotide is called *wild type*, while the least frequent is called *mutant*.

For example, let us consider the following strands of genome coming from three different individuals g_1, g_2 and g_3:

Individual	genome	SNPs	binary representation	genotype
g_1	ATT**A**C**GA**GA**T**	A<u>G</u>AT	0100	0220
	ATT**A**C**CT**GA**T**	AC<u>T</u>T	0010	
g_2	ATT**A**C**CA**GA**T**	ACAT	0000	0002
	ATT**A**C**CA**GA**A**	ACA<u>A</u>	0001	
g_3	ATT**A**C**GA**GA**A**	A<u>G</u>A<u>A</u>	0101	2222
	ATT**T**C**CT**GA**T**	<u>T</u>C<u>T</u>T	1010	
	wild types	ACAT		
	mutant	TGTA		

They all share a common genome, except for the loci marked in boldface, which are SNPs (only two nucleotides occur). The third column highlights the interesting portion of DNA, composed only of SNPs; here, mutant sites are underlined. The last two columns shows the numeric representation of haplotypes and genotypes used in the mathematical model of problem (see Section 2.2). For convenience, the last two rows show wild and mutant nucleotides for all sites.

Technological limitations make it currently impractical to directly collect haplotypes by experimental procedures, but it is possible to collect *genotypes*, i.e., the conflation of a pair of haplotypes, in which the origins of SNPs can not be distinguished. Moreover, instruments can easily identify whether the individual is *homozygous* (i.e., the alleles are the same) or *heterozygous* (i.e., the alleles are different) at a given site. Therefore, haplotypes have to be inferred from genotypes in order to reconstruct the detailed information and trace the precise structure of DNA variations in a population. This process is called *Haplotype Inference* (also known as *haplotype phasing*) and the goal is to find a set of haplotype pairs (i.e., a *phasing*) so that each genotype can be reconstructed by combining a pair of haplotypes from the set.

The main methods to tackle Haplotype Inference are either combinatorial or statistical. Both, however, being of non-experimental nature, need some genetic model that provides some criteria for evaluating the solution returned with respect to actual genetic plausibility. In the case of the combinatorial methods, which are the subject of the present work, one of the most often used criteria is *pure parsimony* [102]. This criterion suggests to search for the smallest collection of distinct haplotypes that solves the Haplotype Inference problem. This criterion is consistent with current observations in natural populations for which the actual number of haplotypes is vastly smaller than the total number of possible haplotypes. The adequacy of this model has already been discussed and motivated elsewhere [56, 100, 102].

Two notable variants of the Haplotype Inference Problem take into account unknown sites and parental relationships between individuals. In this work we only consider the basic version of the problem, in which the full genotype information is available. Moreover we assume that there are no parental relations among population individuals.

The solution method we present is based on a set of AI techniques called metaheuristics (for an introduction to metaheuristics, see, e.g., [31]). In this framework, the Haplotype Inference problem is tackled as a constrained optimization problem with an objective function defined by the phasing evaluation criterion of choice (in this work *pure parsimony*). This chapter and the next are based on previous works, in which different hybrid metaheuristic methods for the Haplotype Inferencewere presented [19, 23, 60]. Moreover, in Chapters 3 and 5 the concepts and algorithms contained in the previously cited works are

extended. Specifically, Chapter 3 addresses two important improvements. As far as experimental validation is concerned, we use a larger and more comprehensive test set including both simulated and real instances of various size and characteristics. All experiments mentioned in [19] have been redone, as well as the parameter configuration of our algorithms and the comparison between the pure and the hybrid variants of the proposed metaheuristics, with longer runtime limits. The best algorithm found in this preliminary evaluation is compared with the state-of-the-art exact solver RPOLY [95], likewise in the previously cited papers. As a further improvement, we also perform an analysis of the instance structure with the aim of determining the instance features which should be taken into account for predicting which of the two approaches (i.e., our hybrid method and RPOLY) will be faster in solving the instance at hand.

Alternative approaches for solving the problem under the pure parsimony hypothesis (HI_{pp}) include heuristic algorithms, such as the simple greedy Clark Rule [54] and an effective modification proposed by Marques-Silva et al. [153], and a recent approach based on graph theoretical interpretation of the problem [232]. As far as exact methods are concerned, we mention the original Integer Linear Programming formulation by Gusfield [102] plus a further variety of ILP models [26, 38, 44, 104, 121, 137]; Semidefinite Programming [114] and SAT models [146, 147, 162]; algorithms based on Answer Set Programming [72] and Pseudo-Boolean Optimization (PBO) [94, 95]. We recommend [45, 93] for comprehensive overviews on combinatorial methods for HI_{pp}.

At present, complete approaches, i.e., the ones that guarantee to return an optimal solution, such as SAT-based ones, are very effective but they seem not to be particularly adequate for large size instances. Hence, the need for approximate algorithms, such as metaheuristics, that trade completeness for efficiency. Moreover, a motivation for studying and applying approximate algorithms is that the criteria used to evaluate the solutions provide an approximation of the actual solution quality, therefore a proof of optimality is not particularly important. To the best of our knowledge, besides [23, 60], the only attempt to employ metaheuristic techniques for HI_{pp} is a Genetic Algorithm [233]. However, also the cited paper does not report results on real size instances.

For the sake of completeness, we mention the major statistical methods that attempt to solve haplotype inference under different genetic models. Widely used algorithms are based on Bayesian approach: Stephens et al. proposed their `PHASE` algorithm [218] and its evolution `PHASE2` [217], Niu et al. introduce another Bayesian algorithm which makes use of Partition Ligation method and Gibbs sampling [164] (see also [216] for a comparison). Scheet and Stephens further elaborate on `PHASE` algorithm in [193] where they present `fastPHASE`, which, as its name suggests, is generally faster than `PHASE` in large datasets. Rastas et al. propose a combination of Bayesian methods with the Context Tree Weighting algorithm [177]. Due to their accuracy, both `PHASE` and `fastPHASE` are typically the term of comparison in works regarding statistical approaches to Haplotype Inference. Other approaches include Hidden Markov Models [131, 176], the Dirichlet Process [243], Deterministic Sequential Monte Carlo [142] and variable order Markov Chains [73, 74, 247].

Finally, we refer to [103], which is an introductory survey about statistical and combinatorial techniques for solving Haplotype Inference under different genetic assumptions, including, of course, parsimony, and the more recent [40], which is primarily focused on statistical methods.

2.2 Mathematical Formulation

In the Haplotype Inference problem we deal with *genotypes*, that is, strings of length m that corresponds to a chromosome with m sites. In particular, not the whole genetic sequence is considered but we focus only on SNP sites.

Each value in the genotype string belongs to the alphabet $\{0, 1, 2\}$. A position in the genotype has value 0 (wild type) or 1 (mutant) if the corresponding chromosome site is a homozygous site (i.e., it has that state on both copies) or the value 2 if the chromosome site is heterozygous. We also call *ambiguous* (resp. *non ambiguous*) those genotypes which have at least a heterozygous site (resp. do not have heterozygous sites). A *haplotype* is a string of length m that corresponds to only one copy of the chromosome (in diploid organisms) and whose positions can assume the symbols 0 or 1.

In this variant of the problem we do not distinguish between the maternal and paternal haplotypes, therefore, given a chromosome, we are interested in finding an unordered pair of haplotypes that can explain the genotype according to the following definition:

Definition 1 (Genotype resolution)**.** Given a chromosome g, we say that the pair of haplotypes $\langle h, k\rangle$ *resolves*, or explains, g, and we write $g = h \oplus k$, if the following conditions hold for all sites $p = 1, \ldots, m$:

$$g_p = 0 \;\Rightarrow\; h_p = 0 \wedge k_p = 0 \tag{2.1a}$$
$$g_p = 1 \;\Rightarrow\; h_p = 1 \wedge k_p = 1 \tag{2.1b}$$
$$g_p = 2 \;\Rightarrow\; (h_p = 0 \wedge k_p = 1) \vee (h_p = 1 \wedge k_p = 0) \tag{2.1c}$$

We also say that h and k are *resolvents* of g.

Conditions (2.1a) and (2.1b) require that both haplotypes must have the same value in all homozygous sites, while condition (2.1c) states that in heterozygous sites the haplotypes must have different values.

Observe that, according to the definition, for a single genotype string the haplotype values at a given site are predetermined in the case of homozygous sites, whereas there is a freedom to choose between two possibilities at heterozygous places. This means that for a genotype string with l heterozygous sites there are 2^{l-1} possible pairs of haplotypes that resolve it. Notice that a genotype with zero or one heterozygous sites is trivially resolved.

As an example, consider the genotype $g = 0212$, then the admissible pairs of haplotypes that resolve it are $\langle 0110, 0011\rangle$ and $\langle 0010, 0111\rangle$.

After these preliminaries we can state the *Haplotype Inference* problem as follows:

Definition 2 (Haplotype Inference problem)**.** Given a population of n individuals, each of them represented by a genotype string g_i of length m, we are interested in finding a *phasing* ϕ, i.e., a function that maps an individual in the population to one pair of haplotypes $\phi : g_i \mapsto= \langle h_i, k_i\rangle$, so that $h_i \oplus k_i = g_i, \forall\ i = 1, \ldots, n$. We call H_ϕ the set of haplotypes used in the construction of ϕ, i.e., $H_\phi = \{h_1, \ldots, h_n, k_1, \ldots, k_n\}$.

2.2.1 The evaluation model

From the mathematical point of view, there are many possibilities for defining the function ϕ (and hence the set H_ϕ), since there is an exponential number of possible haplotypes for each genotype. Therefore, a criterion should be added to the model for evaluating solution quality with respect to some biological model. Such criterion thus measure the biological plausibility of a solution.

A natural model of the Haplotype Inference problem is the already mentioned *pure parsimony* approach that consists in searching for a solution that minimizes the total number of distinct haplotypes used or, in other words,

$$\min |H_\phi| \tag{2.2}$$

A trivial upper bound for $|H_\phi|$ is $2n$ in the case of all genotypes resolved by a pair of distinct haplotypes. Observe however that, in the general case some of the haplotypes in the set possibly resolve more than a single genotype therefore the value $|H_\phi|$ is clearly smaller than $2n$. It has been shown that the Haplotype Inference problem under the pure parsimony hypothesis is APX-hard [137] and therefore NP-hard.

It is important to stress, at this point, that finding a proven optimal solution is not particularly relevant, because the criterion defining the objective function is an approximation of an (unknown) actual quality function. Therefore, approximate approaches that are able to return solutions of a good quality, even if not optimal, are of notable practical importance.

2.2.2 Compatibility and complementarity

Some structural properties of the problem can be captured in a graph $\mathcal{G}_c = (G, E)$, in which the set of vertices coincides with the set of the genotypes and a pair of genotypes g_i, g_j are connected by an edge if they are *compatible*, that is, one or more common haplotypes can resolve both of them. For example, the genotypes 2210 and 1220 are compatible, whereas genotypes 2210 and 1102 are not compatible.

More formally, the property can be defined as follows:

Definition 3 (Genotype compatibility)**.** Let g_i and g_j be two genotypes, g_i and g_j are *compatible* if and only if, for all positions $p = 1, \ldots, m$, the following conditions hold:

$$g_{ip} = 0 \quad \Rightarrow \quad g_{jp} \in \{0, 2\} \tag{2.3a}$$
$$g_{ip} = 1 \quad \Rightarrow \quad g_{jp} \in \{1, 2\} \tag{2.3b}$$
$$g_{ip} = 2 \quad \Rightarrow \quad g_{jp} \in \{0, 1, 2\} \tag{2.3c}$$

We express the compatibility relation by writing $g_i \sim g_j$.

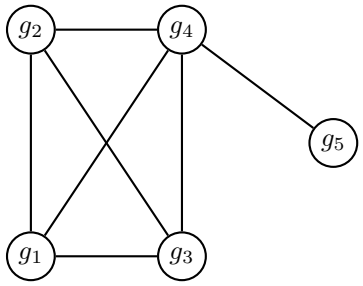
It can be verified easily that the compatibility relation is reflexive and symmetric, but not transitive. For example, let $g_1 = 1022$, $g_2 = 2021$ and $g_3 = 0021$; we have $g_1 \sim g_2$ and $g_2 \sim g_3$ but $g_1 \not\sim g_3$. These two properties entail that the compatibility relation can be modeled by a simple undirected graph $\mathcal{G}_c$ whose adjacency matrix is a $(0, 1)$-symmetric matrix. If we disregard self-loops, the adjacency matrix has all zeros on its main diagonal.

An example compatibility graph can be found in Figure 2.1.

A similar property can be also defined between a genotype and a haplotype:

$g_1 : (2210212)$
$g_2 : (2112110)$
$g_3 : (1212122)$
$g_4 : (1222122)$
$g_5 : (1202201)$

(a) A set of 5 genotypes



(b) The corresponding compatibility graph

Figure 2.1: An example of compatibility graph for a set of genotypes.

Definition 4 (Genotype and haplotype compatibility)**.** Let g be a genotype and h a haplotype, g and h are *compatible*, and we write $g \sim h$, if, for all positions $p = 1, \ldots, m$, the following conditions hold:

$$g_p = 0 \quad \Rightarrow \quad h_p = 0 \tag{2.4a}$$
$$g_p = 1 \quad \Rightarrow \quad h_p = 1 \tag{2.4b}$$
$$g_p = 2 \quad \Rightarrow \quad h_p \in \{0, 1\} \tag{2.4c}$$

In such case we can also write that h *explains* g.

Another useful property, which immediately comes from the resolution definition, is the one of *complementarity*:

Proposition 1 (Haplotype complement)**.** *Given a genotype g and a haplotype $g \sim h$, there exists a unique haplotype k such that $h \oplus k = g$. The haplotype k is called the* complement *of h with respect to g and is denoted with $k = g \ominus h$.*

This important property is the basis of Clark's Rule [54] and it is also exploited in the algorithms presented in this work. As an example, considering again the genotype $g = 0212$ and the haplotype $h = 0110$, the only haplotype k such that $g = h \oplus k$ is 0011.

Some notable properties can be inferred by the compatibility graph. One of the most important is the lower bound on the number of haplotypes required. The availability of a lower bound is of primary importance in the context of exact methods, such as the ones cited previously, or in other techniques that make use of bound information.

The lower bound presented in [148] involves finding the maximum independent set of a graph.

Proposition 2 (Lower Bound)**.** *Let $\mathcal{G}_c$ be the compatibility graph and M be the maximum independent set in $\mathcal{G}_c$; a lower bound for the Haplotype Inference under parsimony is:*

$$2|M| - \sigma$$

where σ is the number of non ambiguous genotypes.

This bound is easily explained. The maximum independent set M in the compatibility graph $\mathcal{G}_c$ is the largest set of those genotypes which are *not* compatible to each other, therefore they cannot share a haplotype in their resolutions. It follows that, at least $2|M|$ haplotypes are required, minus one for each non ambiguous genotype, since non ambiguous genotypes have no heterozygous site.

Another important general property that can be exploited, especially in a preprocessing phase, is the identification of connected components in the compatibility graph. Let us call $C = \{G_1, G_2, \ldots, G_N\}$ the connected components in $\mathcal{G}_c$; the union $\bigcup_{c \in C} c$ equals the genotype set G of an instance, so C is a partition of G. Each connected component can be solved as a separate instance because it does not share haplotypes with other connected components. If we denote with $H_\phi(G_i)$ $(i = 1, \ldots, N)$ the haplotype sets of a phasing for the i-th sub-instance, $\bigcup_{c \in C} H_\phi(c)$ is the haplotype set of a feasible phasing for the whole instance G. A special case of independent instance is represented by isolated nodes, i.e., genotypes that are not compatible with any other genotype. Any feasible resolution for such genotypes is equivalent in terms of the Maximum Parsimony criterion.

The utility of this property extends of course to any technique, beside metaheuristics, and will be used as a preprocessing step in all inference algorithms presented in this thesis.

These concepts allow to avoid unnecessary checks in the determination of the resolvents of a given genotype and, as we will see in the following sections, the graph structure can be exploited to improve the knowledge on the problem. Moreover, as we will point out in Section 3.4, the structure of the compatibility graph is one determinant of the hardness of an instance.

Other useful information obtained by the compatibility graph are described in Chapter 5.

2.3 Notable variants of the Haplotype Inference

In this last section we present some notable variants of Haplotype Inference that can be found in the literature. From a biological point of view these alternative formulations address real-world cases that do occur in practice. For some formulations, the algorithms presented in this dissertation can be naturally extended.

Parental information (Trios). When association studies are conducted, the population taken under examinations often contains familial units [39, 119, 151]. In such cases such parental constraint must be taken into account in the problem formulation. In the typical scenario, a population of specimen may contain one or more *trios*, that is, triplets of individuals composed of two parents and a child. In a trio, the child's haplotypes must come from each parent. This additional constraint makes haplotype inference easier.

For instance, suppose we have a "mother" genotype $g_m = 022010$, a "father" genotype $g_f = 012202$ and a "child" genotype $g_c = 021220$. Here is how this additional information can be used. The child can be resolved by the following

pair of haplotypes $\langle 0*1**0, 0*1**0\rangle$, where $*$'s denote unknown values. If we propagate this information to the mother we have that a possible resolution is $\langle \underline{0*1010}, 0*0010\rangle$ (we underline the haplotype shared with the child); if we propagate this information to the father we have the following resolution template $\langle \underline{011*00}, 010*01\rangle$. Finally, we can infer that the only possible resolution for the child is $\langle 001010, 011100\rangle$, for the mother is $\langle \underline{001010}, 010010\rangle$ and for the father is $\langle \underline{011100}, 010001\rangle$. If we had not parental information, we would not be able to correctly phase such small instance.

It can be easily seen that, if three genotypes are involved in a trio, they form a triangle in the compatibility graph.

Missing information (Unknowns). Another typical real-world challenge is to cope with missing data [193, 217]. Sometimes the precise genotype information might not be available for all sites. In such cases the usual formulation of Haplotype Inference is modified to take into account *unknown sites* (or *unknowns* for short). Instead of dealing with ternary genotype string, Haplotype Inference with Unknown Data is concerned with genotype strings in a quaternary alphabet, such as $\{0, 1, 2, 9\}$, where 9 denotes the unknown site. The introduction of this new concept changes the fundamental properties of Haplotype Inference, the most important of which is the compatibility relation. Basically, a 9 act as a wildcard in the sense that any pair of alleles can be used to solve an unknown site. For instance, a genotype 01090 can be resolved by any of the following pairs: $\langle 01000, 01000\rangle$, $\langle 01000, 01010\rangle$, $\langle 01010, 01000\rangle$, $\langle 01010, 01010\rangle$. In general, a unknown site allows for more degrees of freedom with respect to heterozygous sites. The most remarkable difference introduced by this variant is that the complement of haplotype h with respect to a genotype g *is not unique* if g contains unknowns. For example, the complements of 01011 with respect to 21092 are both 11000 and 11012.

Minimum entropy. This interesting variant is based on a different genetic model than pure parsimony [100]. Haplotype Inference by Minimum Entropy, a specialization of the more general Minimum-Entropy Set Covering Problem [105], substitutes Equation 2.2 with another formula based on Shannon entropy. Let G be a Haplotype Inference instance and ϕ a feasible phasing; we define a function *coverage*, denoted $cvg(h, \phi)$, as $cvg(h, \phi) = \sum_{g \in G} c_{g,h}$, where the contribute $c_{g,h}$ is:

$$c_{g,h} = \begin{cases} 0 & \text{if} \quad h \notin \phi(g) \\ 1 & \text{if} \quad \phi(g) = \langle h, k\rangle \vee h \neq k \\ 2 & \text{if} \quad \phi(g) = \langle h, h\rangle \end{cases}$$

Basically, the coverage function counts the occurrences, with their multiplicities, of each haplotype in a phasing. Notice also that $\sum_{h \in H_\phi} cvg(h, \phi) = 2|G|$. The entropy of a phasing is thus:

$$\mathcal{H}(\phi) = -\sum_{h \in H_\phi} \frac{cvg(h, \phi)}{2|G|} \log \frac{cvg(h, \phi)}{2|G|}$$

where we can interpret the quantity $\dfrac{cvg(h, \phi)}{2|G|}$ as the "probability" of haplotype h.

Haplotype Inference by Minimum Entropy has been demonstrated to be $\mathcal{NP}$-hard [105].

Chapter 3

Hybrid Metaheuristics for Haplotype Inference

In this chapter, and in the following one, two metaheuristic techniques for tackling the Haplotype Inference problem are described. The first of such techniques is an hybrid algorithm that integrates Ant Colony Optimization, a population based metaheuristic, with Tabu search, a stochastic local search technique. The second one (Chapter 4) is an instance of a Master-Slave Genetic Algorithm, a particular variation on the usual genetic algorithm framework, which borrows its main idea from techniques such as variable fixing and large neighborhood search.

Both algorithms draw inspiration by the so called *Clark's Rule*, a deterministic greedy constructive procedure, which, based on complementarity (Property 1), tries to construct a feasible phasing for a problem instance. The underlying idea in both algorithms is to exploit the high-level learning mechanism of metaheuristics to guide the choices that Clark's Rule has to make when solving an instance. This simple but important heuristic will be described in more detail in the following section.

3.1 Clark's Rule

Clark's Rule is a very simple deterministic greedy heuristic first introduced in [54]. This heuristic iteratively constructs a set H of haplotypes (a phasing can be then easily obtained) given an ordering of the genotypes by repeated application of the following inference rule: suppose that an unresolved genotype g is compatible with a haplotype h in the current set H, add $g \ominus h$ to H and remove g from the list of genotypes to visit.

It is important to notice that, despite Clark did not explicitly address the haplotyping problem by maximum parsimony, Clark's Rule intrinsically follows a parsimonious criterion because on each iteration it tries to re-utilize haplotypes.

This heuristic faces important problems. First of all, it needs a non-empty haplotype set to bootstrap the search procedure. Such initial set is composed by unambiguous genotypes—since they are identical to haplotypes—and those haplotypes that resolve genotypes with only one heterozygous site—since their

resolution is unique. It is not always possible, though, to obtain an initial set because an instance might not contain such trivially solvable genotypes and, even if it did, the haplotypes in the initial set must be compatible to at least one other genotype in the instance, otherwise the procedure could not go any further. For example, consider the instance $G = \{01210, 01012, 20120, 20112, 20212\}$; the only bootstrap set, obtained by the first two genotypes, is $H_0 = \{01010, 01110, 01011\}$ whose haplotypes, unfortunately, are not compatible to the remaining genotypes. Following a similar argument, it can be concluded that Clark's Rule does not always manage to return a resolution for *all* genotypes. In these cases Clark's Rule could indeed be randomly seeded with a suitable haplotype; for instance, in the previous example, we could add haplotype 00100 and let the procedure continue.

Another problem is met whenever the partial solution H contains more than one candidate resolvent for the current genotype as different choices at an early stage would constrain further choices.

In the last possibility, Clark implements a tie-breaking algorithm dependent on the genotype visiting order. Nevertheless, beside the bootstrapping problem, the correctness of the algorithm is not guaranteed and is, anyway, deeply dependent on the supplied genotype ordering. In its work, Clark addresses the dependence on the visiting order by running Clark's Rule multiple times, each time with a different ordering of genotypes [1], and keeping the solution with the greatest number of resolved genotypes.

The problem of correctness is also analysed in [101] by Gusfield. In its work Gusfield formulates the Maximum Resolution problem which is defined as follows: given a genotype set, find an ordering that, when supplied to Clark's Rule, maximises the number of resolved genotypes. Of course, if we could answer such question we could also determine if there is a visiting order that resolves all genotypes in an instance. Unfortunately, Gusfield also demonstrates that the Maximum Resolution problem is $\mathcal{NP}$-hard.

Before we move on with the description of the ACO algorithm, it is worth to notice that, with the due adjustments, Clark's Rule *could* indeed become a correct algorithm. These modifications have been already mentioned in the text and all involve the introduction of non-determinism in the execution of the algorithm. A similar path has been followed by Clark: in its multistart algorithm, he essentially introduces the same amount of non-determinism, since the genotype ordering are generated in an uninformed random manner.

There are basically three cases in which the Rule fails:

1. no haplotype in the current partial solution H can resolve an unvisited genotype (the same case is true if no bootstrap set could be found);

2. there are more than one haplotypes in H that could solve the current genotype (we disregard the heuristic dependant on genotype order that Clark introduced);

3. at any iteration, there may be more than one genotype solvable by some haplotype in the current set H.

Suggested modifications, for each case, could be respectively:

[1] In the metaheuristic terminology, we can say he used a random multistart.

1. randomly pick an unvisited genotype g and a haplotype $h \sim g$; add h and $g \ominus h$ to H;

2. randomly select a haplotype from the set $\{h \mid g \sim h\}$ where g is the current genotype;

3. randomly select an unvisited genotype g such that $\exists h \in H, g \sim h$.

Such stochastic version of Clark's Rule can form the basis for more effective haplotyping algorithms. The main intuition behind the algorithms presented in this chapter and the next one can be summarised by the following statement: in principle, an optimal solution could be found if an oracle were to indicate, for each one of the cases above, the right choices. This is in general not possible, but an iterative and adaptive search strategy could be very effective in exploring these possibilities.

3.2 Metaheuristic techniques for Haplotype Inference

In previous preliminary works the problem was tackled by means of a single metaheuristic technique [23, 60].

Although the choice of a genetic model only involves the redefinition of the objective function, hence the quality function (see Section 3.2.1), this algorithm is structurally biased towards the pure parsimony approach, because the lower level genotype resolution, essentially a modified version of Clark's Rule which exploit the search experience, aims to re-use already selected haplotypes as we will see further in the text.

3.2.1 Ant Colony Optimization

The Haplotype Inference problem definition makes constructive procedures very promising. Indeed, a constructive procedure can incrementally build a set H of haplotypes which, taken in pairs, resolve the genotypes. Such a procedure can start from an empty set and add one or two haplotypes at a time, while it scans the set of genotypes G. The objective is to build H as small as possible, i.e., to find a minimal cardinality set of haplotypes that composes the phasing. To this aim, new haplotypes should be added to H only when necessary, i.e., when no pair of haplotypes already in H resolves the current genotype g. Essentially, the algorithm, at its core, is a modified version of the basic Clark's Rule which tries to deal with all the sources of non-determinism listed previously by exploiting the learning component of ACO.

In [23], an ant-based algorithm has been proposed, which follows the well-renowned Ant Colony Optimization metaheuristic [65].

ACO-HI (Algorithm 3.1) is a stochastic constructive procedure that operates on two levels: on the higher level an ACO for finding a good visiting order of genotypes is run while on the lower level, another ACO algorithm searches for the haplotypes to be added to H_ϕ. The two levels are of course coupled, as the order in which genotypes are considered is influenced by the current set of

```
1: procedure ACO-HI
2: Preprocessing phase
3: while terminating conditions not met do
4:   for all a ∈ A do
5:     while not all genotypes are resolved do
6:       g ← chooseNode(G)
7:       resolve genotype g
8:     end while
9:   end for
10:  pheromone update
11: end while
```

Figure 3.1: ACO-HI algorithm

haplotypes in H_ϕ and, conversely, a generic step in the construction of H_ϕ depends on the previously resolved genotypes. In our implementation, termination conditions can be a runtime limit or a maximum number of iterations.

Before applying the two-level ACO, the problem instance is preprocessed by a procedure that eliminates replicates among genotypes and identifies disconnected parts in the compatibility graph that can then be treated as independent instances as described at the end of Section 2.2.2.

Lower level: genotype resolution.

An ant a builds a solution by considering in turn each genotype $g \in \mathcal{G}_c$ (the order is defined in the higher-level), where $\mathcal{G}_c$ is the compatibility graph for the instance, and finding a resolution for it (Line 7 in Algorithm 3.1). When a new resolvent has to be added to H_ϕ, the values of its heterozygous sites, either 0 or 1, are chosen on the basis of pheromone values [2] and a heuristic strategy.

This strategy constrains the possible choices of a value and aims at minimizing the number of haplotypes to be added to the solution, for example by forcing the new resolvent to be compatible with an unvisited neighbor of g in the compatibility graph. In those sites unconstrained by the heuristic strategy, we have a pheromone guided binary choice: the value is chosen with a probability proportional to its pheromone value.

Excluding the case in which H_ϕ already contains a pair of haplotypes resolving g, the are three different cases to be considered for the resolution of a genotype g in this step of the algorithm: *(i)* no resolving candidates in H_ϕ, i.e., $\nexists h \in H_\phi \mid h \sim g$; *(ii)* one candidate; and *(iii)* more than one candidate. In the following, we detail the procedure defined for these cases:

Case (i): A haplotype h is built by a pheromone guided construction procedure, as previously described. Then, $k = g \ominus h$, the complement of h, is calculated and both are added to H_ϕ.

Case (ii): When one resolving candidate is already available, its complement w.r.t. g is built and this step completed as in the previous case.

[2] Homozygous sites do not represent choice points as they are directly assigned because the haplotype we are constructing must resolve g.

Case (iii): When there are two or more candidates in the partial solution H_ϕ that can resolve g (but no pair of them can resolve it), we choose one among these haplotypes by iteratively considering each (heterozygous) site and applying the following procedure: if, among the candidates, the homologous sites have different values (i.e., at least in a pair there are both values 0 and 1) one of the two is chosen probabilistically (using pheromone values) and all the candidates with a different value are discarded. The procedure ends when only one candidate, call it h, is left. Finally $g \ominus h$ is added to H_ϕ.

ACO-HI algorithm can be further improved by slightly modifying the procedure implemented for case i. In fact, since the new haplotype added to H_ϕ must resolve the current genotype g, a heuristic bias toward the construction of a haplotype that also resolves another genotype compatible with g can be beneficial. Thus, the genotype g' that has to be visited after g is determined by the higher level and a haplotype is probabilistically constructed (as in the original procedure) that not only resolves g, but also g'. In this way, haplotype construction is not only guided by pheromone but also by a heuristic that avoids building a new haplotype compatible only with genotype g. We will refer to the improved version of ACO-HI as ACO-HI$^+$. In Section 5.2 we will examine limitations of this selection heuristic and a possible extension.

Higher level: genotypes visiting order.

The order in which genotypes are visited has a strong influence on solution quality, therefore the higher level of the algorithm tries to learn a good genotype visiting order. This learning mechanism is primarily guided by pheromone associated to the edges of the compatibility graph. In this way, pairs of consecutive genotypes in the series are learned. It would be possible to learn larger building blocks, such as triplets, but we decided to limit the case to pairs because of efficiency reasons.

Formally, every edge (g_i, g_j) of the compatibility graph is associated to a pheromone value τ_{ij} and the probability to move from node g_i to node g_j is given by:

$$p(g_i, g_j) = \begin{cases} \frac{\tau_{ij}}{\sum_{l \in adj(g_l)} \tau_{il}}, & \text{if } g_j \in adj(g_i) \\ \frac{1}{|U|}, & \text{if } g_j \in U \wedge adj(g_i) = \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

where $adj(g_i)$ is the set of nodes adjacent to g_i (i.e., the compatible genotypes) still unresolved, and U is the set of currently unvisited genotypes not compatible with genotype corresponding to g_i. In such a way, if g_i has adjacent unresolved nodes, then one among them is chosen according to pheromone values; otherwise, the next genotype in the sequence is chosen randomly among the remaining unresolved genotypes.

Pheromone update.

The objective function of the problem is the cardinality of H_ϕ, that has to be minimized. Therefore, as a quality function used for the reinforcement, we chose the function $F(H_\phi) = 2n - |H_\phi|$. Pheromone is updated in the two levels with

the same evaporation parameter and quality function. The only difference is that the solution components of the higher level are edges of the compatibility graph, while in the lower level they are nodes representing values to assign to haplotype sites.

3.2.2 Stochastic Local Search

In this section we describe the local search method used in our hybrid algorithm.

In [60] a stochastic local search method was proposed based on a simple neighborhood definition and on a heuristic reduction procedure. The neighborhood considered was built on the *1-Hamming* move, which simply swaps the values assigned to a given site between the two haplotypes currently resolving a given genotype. Even though it has shown to be quite effective, that method presented some limits, mainly related to the fine level of granularity of the neighborhood. Since then, the local search method has been improved by empowering it with a more effective neighborhood definition which also exploits some knowledge about the problem structure. This variant of the algorithm [61] makes use of a move definition that employs a haplotype already included in the set H_ϕ also for resolving other compatible genotypes.

The local search model is defined by specifying the following three entities: the *search space*, the *cost function* and the *neighborhood relation*, which will be detailed in the following.

Search space and cost function

As for the search space, we consider states composed by the resolutions $g = h \oplus k$, for all $g \in G$. Therefore in this representation all the genotypes are fully resolved at each state by construction. Thus, the search space is the collection of sets ϕ defined as in the problem statement.

The cost function is a measure of solution quality including components related both to the solution evaluation criteria of choice and to heuristic information that could guide search toward good solutions.

The component related to the evaluation criterion is an objective function defined as the cardinality $|H_\phi|$ of the set of haplotypes employed in the resolution; in formulae:

$$f_1(\phi) = |H_\phi| \tag{3.2}$$

Moreover, we also include a heuristic measure related to the potential quality of the solution. In this respect, we counted the number of incompatible sites between each genotype/haplotype pair and the component of the cost function is expressed by the following formula:

$$f_2 = \sum_{h \in H} \sum_{g \in G} \sum_{j=1}^{m} 1 - \chi(h[j] \sim_c g[j]) \tag{3.3}$$

In the formula, χ denotes the truth indicator function, whose value is 1 when the proposition in parentheses is true and 0 otherwise.

The cost function F is then the weighted sum of the two components:

$$F = \alpha_1 f_1 + \alpha_2 f_2 \tag{3.4}$$

1: **procedure** TS()
2: $s \leftarrow$ GenerateRandomInitialSolution()
3: $s \leftarrow$ Reduce(s)
4: $s_b \leftarrow s$
5: **while** termination conditions not met **do**
6: $\mathcal{N}_a(s) \leftarrow \{s' \in \mathcal{N}(s) \mid s' \text{ does not violate the tabu condition}\}$
7: $s' \leftarrow \mathsf{argmin}\{F(s'') \mid s'' \in \mathcal{N}_a(s)\}$
8: $s \leftarrow s'$ {i.e., s' replaces s}
9: **if** $F(s) < F(s_b)$ **then**
10: $s_b \leftarrow s$
11: **end if**
12: **end while**
13: **return** best solution found s_b

Figure 3.2: High level scheme of Tabu Search for Haplotype Inference

in which the weights α_1 and α_2 must be chosen for the problem at hand to reflect the trade-offs between the different components. In our experimentation we chose the values $\alpha_1 = \alpha_2 = 1$.

Neighborhood relation and search strategy

We designed a stochastic local search technique, based on the *Tabu Search* metaheuristic template. The strategy is defined in Figure 3.2. The algorithm starts with a set of randomly generated haplotypes of cardinality at most $2n$, where n is the number of genotypes. Then, a reduction procedure is called whose aim is to reduce the number of haplotypes by exploiting the structure of the compatibility graph. This procedure was first presented in [60] and tries to remove from H_ϕ haplotypes that are not necessary to resolve some genotype.

After this preprocessing phase, the solver explores the search space by iteratively modifying pairs of resolving haplotypes trying to reduce the value of the cost function F. Then, the iterative process is repeated as long as a termination criterion is met: in our implementation, we allotted either a maximum runtime or a combination of maximum iterations and maximum number of iterations without improvement (idle iterations). Tabu Search explores all the neighbors of the incumbent solution s that can be reached by applying moves that are not in the tabu list and chooses the best neighbor (Lines 6–8). A move is tabu if it or its inverse have been applied in the last tl iterations.

3.2.3 The hybrid algorithm

The structure of the hybrid algorithm is shown in Figure 3.3. Its structure is relatively simple and consists in a sequential invocation of ACO and a variant of TS, denoted by TS', which employs the state returned by the ant-based algorithm as the initial state of the Tabu Search (in other words, TS with Line 2 omitted).

We also experimented with a restarting strategy, i.e., putting ACO▷TS in a loop and executing several searches from scratch, but we experimentally discovered that its performances were in the same slot as the simple strategy presented above.

```
1: procedure ACO▷TS()
2: s ← ACO()
3: s ← TS'(s)
4: return s
```

Figure 3.3: The ACO▷TS hybrid algorithm

Benchmark set	N. of instances	N. of genotypes	N. of sites
Harrower Uniform	200	10÷100	30÷50
Harrower Hapmap	24	5÷68	30÷75
Marchini SU1/SU2/SU3	100	90	171 ÷ 187
Marchini SU-100kb	29	90	18
ABCD	10	5 ÷ 50	27
ACE	10	5 ÷ 50	52
IBD	17	4 ÷ 50	103
Extra	6	50, 100	75, 100
Eronen	17	500	32
Climer	7	39, 80	5 ÷ 47
Daly	2	147, 387	103

Table 3.1: Main features of the benchmarks.

3.3 Experimental analysis

In this section we show the advantage of the hybrid algorithm with respect to its basic components and we compare it with RPOLY, the state-of-the-art exact solver for the problem.

The algorithms have been developed in C++, exploiting the EasyLocal++ framework for the local search component [62]. The software was compiled with the GNU g++ compiler v. 3.4.6 and tested on cluster composed of Intel Xeon 3.0GHz machines running Linux (kernel 2.6.18). All software has been compiled with -O3 flag on.

In the following experiments we employ thirteen sets of instances, whose features are summarized in Table 3.1:

- Harrower Uniformand Harrower Hapmapare taken from [38];
- Marchini SU1, Marchini SU2, Marchini SU3 and Marchini SU-100kb are downloadable from http://www.stats.ox.ac.uk/~marchini/phaseoff.html;
- ABCD, ACE, IBD, Extra are a selection of instances from their respective instance set which have been provided by the same authors as [94];
- Eronen is a selection of instances from the dataset used in [73], which we refer to for a complete description;
- Climer: this is the dataset labeled *"Known haplotype data"* from [56];
- Daly contains two instances: one is the original dataset from [58] (downloadable at: http://www.broadinstitute.org/archive/humgen/IBD5/

`haplodata.html`) the other is the one used in [73]; in both we considered all missing sites as heterozygous.

All complete datasets are available at this URL[3]: `http://apice.unibo.it/xwiki/bin/view/HaplotypeInference/`.

In order to select the parameters for the solvers we proceeded as follows. Out of the thirteen instance set, we first identified a subset as our training set. In this stage, we adopted as a measure of hardness of an instance the runtime of an algorithm, given a fixed amount of maximum and idle iterations. From preliminary sample runs, we concluded that Harrower Uniform and Harrower Hapmap were representatives of "easy" instances while Marchini SU1, SU2, SU3 and SU-100kb were representatives of "hard" instances. This rough classification is indeed confirmed by looking at Table 3.3. We then made 10 independent runs of ACO▷TS, providing 1000 idle iterations and 2000 total iterations limits (shared between ACO-HI$^+$ and TS components) and no timeout. For each instance set, we recorded the longest execution time. Next we run the other two pure algorithms (ACO-HI$^+$ and TS) by setting as timeout for each instance in a given set the aforementioned longest execution time; no limits on the number of idle and maximum iterations were imposed. This way we could fairly compare the three algorithms.

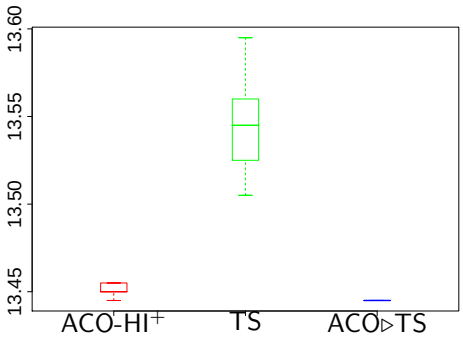
The other parameters were kept fixed throughout the evaluation process: 0.1 evaporation factor, 75 artificial ants (for ACO-HI$^+$) and a variable length tabulist between 20 and 30 (for TS). Also, as initial solution for the TS algorithm, we chose the one returned by the ACO algorithm after a single iteration. These parameters have been determined after a *full factorial* analysis on a reduced set of instances.

3.3.1 Analysis of ACO-HI$^+$, TS and ACO▷TS

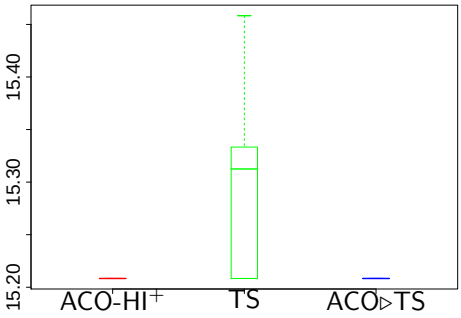
The results of the analysis of the ACO-HI$^+$, TS and ACO▷TS algorithms are presented in Figure 3.4 where we compare, for each training dataset, the performance of our algorithms. The boxplots [83] drawn in the graphics were calculated as follows: 1) for a dataset S and an algorithm a, we made 10 independent runs of a on every instance $s \in S$ and recorded the solution value (as per Equation 2.2, to be minimized); 2) this way, we obtained a performance matrix $M_a = d \times 10$ matrix, where d is the cardinality of the dataset; 3) we then computed the average on each column of M_a, i.e., we took the average across all instances, and obtained a vector of 10 data points v_a. We repeated this procedure for each algorithm $a \in \{\text{ACO-HI}^+, \text{TS}, \text{ACO}\triangleright\text{TS}\}$ and each training dataset; each boxplot refers to each one of the vector v_a calculated at step 3. We notice that the average took at step 3 does not bias the results because all instances in a datasets are comparable with respect to solution value.

Table 3.2 and 3.3 report average and standard deviation of the cumulative running time for solving all the instances of each set and on the time for each of the algorithm to find the best solution. From Table 3.3 it can be seen that, on average, Marchini's datasets are actually more computationally intensive, while Harrower's datasets do not require a similar computational effort since the algorithms have shorter runtimes. Table 3.4 similarly summarizes the *efficiency*

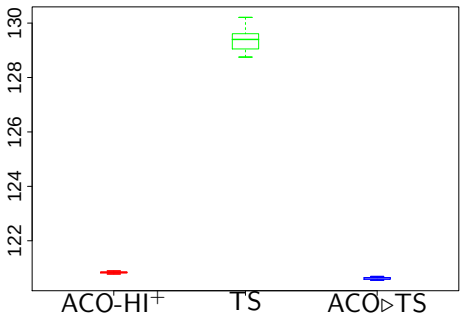
[3]We thank Ana Sofia Graça and Inês Lynce for kindly providing us their instances and solvers; we also thank Ian M. Harrower and Sharlee Climer for sending us their datasets.



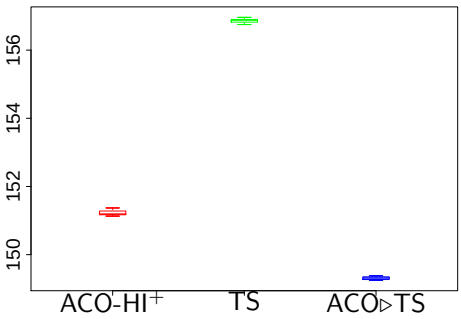
(a) Harrower Uniform



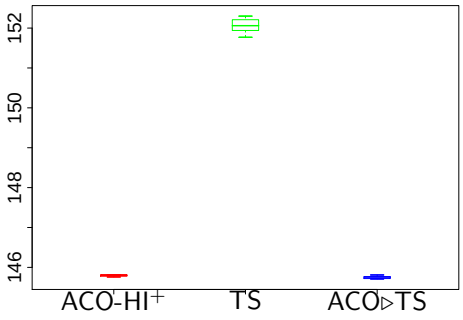
(b) Harrower Hapmap



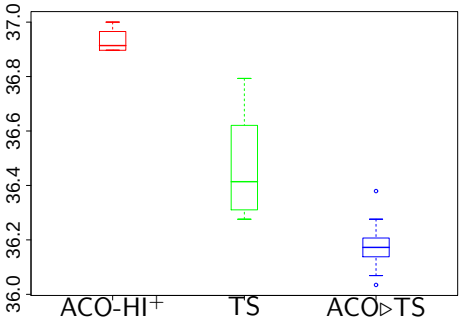
(c) Marchini SU1



(d) Marchini SU2



(e) Marchini SU3



(f) Marchini SU-100kb

Figure 3.4: Solution value found by ACO-HI$^+$, TS and ACO▷TS (on the y-axis is reported the *average* number of haplotypes over all the instance set).

of each algorithm, measured as the ratio—expressed in percentage—between the time to find the best solution and total running time.

Data in Figure 3.4 clearly show that ACO▷TS is almost always superior to the plain local search and statistically indistinguishable from ACO-HI$^+$, except in one single instance set, namely Marchini SU-100kb, while ACO▷TS systematically dominates the other algorithms.

The results also demonstrate that the hybrid algorithm, and the pure versions as well, is very stable with respect to solution value, that is, despite ACO▷TS being a stochastic algorithm, different realizations have a high probability to return solutions whose values fall in a small interval.

Experimental results show that even the proposed coarse integration of the two basic algorithms improves solution quality. This can be explained by the fact that, in general, population algorithms are good at finding promising areas in the search space, rapidly improving the solution in the early stages of the search. On the opposite side, they are not able to explore promising areas, that is, intensifying around good solutions, which, instead, is one of the characteristics of local search algorithms. This is also witnessed by the different efficiency figures reported in Table 3.4. Moreover it can be stressed that, with this set of parameters, the hybrid algorithm compares favorably against the pure ACO for what running times are concerned, in fact, not only it is faster, but also has comparable time-to-solution and a higher efficiency.

Marchini SU-100kb instance set exhibit an unusual phenomenon: in those instances the ACO-HI$^+$ algorithm is dominated even by the pure local search. This behavior is due to the particular structure of the instances—we conjecture that is due to the sparseness of their compatibility graph.

To conclude, this is an example that shows how much a hybrid approach could be particularly useful in general, since each component compensates for each other weaknesses.

3.3.2 Comparison between ACO▷TS and rpoly

In the second phase of the experimentation process we compare ACO▷TS with the latest version of RPOLY, namely 1.2.1 at the time of this writing, the state-of-the-art exact method for the problem. RPOLY [94] is a Pseudo Boolean Optimisation (PBO) model of the Haplotype Inference by Pure Parsimony that is based on the integer programming formulation PolyIP [37]. PolyIP is a *0-1 integer programming* formulation (i.e., variables have domain $\{0, 1\}$ and all constraints and the objective function are linear with integer coefficient), therefore is automatically a PBO model. What RPOLY basically does is to take the PolyIP model of an instance and optimise it by reducing the formulation, adding symmetry breaking constraints and also integrating a lower bound [96]. The resulting model is finally passed to a standard PBO solver (MiniSAT+ [69] in the current implementation). In addition, since its latest release, RPOLY can be run as an *incomplete anytime solver* by setting a runtime limit: in the allotted time span, either it finds the optimum or it returns an approximated solution.

Like many problems in the field of bioinformatics, Haplotype Inference is not time-critical: high accuracy and solution quality is much more valuable than a fast-computed solution. That said, we decided to give the competing algorithms a large amount of time to solve each instance.

	ACO-HI$^+$		TS		ACO▷TS	
Instance set	**running time**	**time to best**	**running time**	**time to best**	**running time**	**time to best**
Marchini SU1	8835.5 (101.943)	1609.9 (191.044)	1182.2 (20.069)	1181.0 (20.020)	5667.6 (88.565)	1693.3 (99.522)
Marchini SU2	4046.1 (35.478)	733.6 (56.826)	21.4 (3.040)	21.0 (2.933)	3258.3 (25.679)	572.6 (40.098)
Marchini SU3	7587.9 (122.346)	1208.0 (165.636)	715.6 (45.844)	714.8 (46.151)	4529.3 (79.467)	1268.0 (100.410)
Marchini SU-100kb	4954.6 (107.687)	923.9 (245.572)	1331.1 (118.686)	1331.1 (118.686)	3142.7 (93.112)	1059.5 (93.865)
Harrower Uniform	8864.6 (11.191)	232.5 (29.817)	800.9 (8.514)	800.7 (8.521)	4788.4 (30.345)	552.0 (34.880)
Harrower Hapmap	742.7 (2.283)	25.4 (26.834)	178.1 (27.142)	178.1 (27.142)	449.8 (15.542)	89.7 (15.324)

Table 3.2: Cumulative running times and cumulative times to the best solution found by the algorithms (in seconds of CPU time).

	ACO-HI$^+$				TS				ACO▷TS			
Instance set	**running time**		**time to best**		**running time**		**time to best**		**running time**		**time to best**	
Marchini SU1	88.4	(1.019)	16.1	(1.910)	11.8	(0.201)	11.8	(0.200)	56.7	(0.886)	16.9	(0.995)
Marchini SU2	40.5	(0.355)	7.3	(0.568)	0.2	(0.030)	0.2	(0.029)	32.6	(0.257)	5.7	(0.401)
Marchini SU3	75.9	(1.223)	12.1	(1.656)	7.2	(0.458)	7.1	(0.462)	45.3	(0.795)	12.7	(1.004)
Marchini SU-100kb	170.8	(3.713)	31.9	(8.468)	45.9	(4.093)	45.9	(4.093)	108.4	(3.211)	36.5	(3.237)
Harrower Uniform	44.3	(0.056)	1.2	(0.149)	4.0	(0.043)	4.0	(0.043)	23.9	(0.152)	2.8	(0.174)
Harrower Hapmap	30.9	(0.095)	1.1	(1.118)	7.4	(1.131)	7.4	(1.131)	18.7	(0.648)	3.7	(0.638)

Table 3.3: Cumulative running times and cumulative times to the best solution found by the algorithms (in seconds of CPU time) averaged on the number of instances per set.

Instance set	ACO-HI$^+$		TS		ACO▷TS	
Harrower Uniform	2.6%	(0.3%)	99.7%	(0.009%)	11.5%	(0.7%)
Harrower Hapmap	3.4%	(3.6%)	97.1%	(0.3%)	19.9%	(2.7%)
Marchini SU1	18.2%	(2.0%)	99.7%	(0.009%)	29.9%	(1.3%)
Marchini SU2	18.1%	(1.4%)	97.1%	(0.3%)	17.6%	(1.1%)
Marchini SU3	15.9%	(1.9%)	99.5%	(0.03%)	28.0%	(1.7%)
Marchini SU-100kb	18.6%	(4.5%)	99.9%	(0.002%)	33.7%	(2.0%)

Table 3.4: Efficiency of the algorithms (in percentage).

As for the experimental setup of RPOLY, we set a 16 hours timeout for each instance, amount determined after sample runs on the whole benchmark. The parameters of the hybrid algorithm were set as in the previous training experiments, and we assigned a maximum runtime of 96 minutes.

In order to fairly compare ACO▷TS against the complete algorithm, we run it 10 times on each instance and then we took the best solution. To sum up, we compared the anytime version of RPOLY with a 16 hours timeout against the best solution returned by our ACO▷TS in 10 runs of 96 minutes each. In the end, both algorithms have a runtime budget of 16 hours for each instance. Results on these experiments are shown in the scatter plots pictured in Figures 3.5, 3.6, 3.7; the solution quality returned by ACO▷TS is plotted on the y-axis while in the x-axis is plotted the solution value returned by RPOLY. This way, a dot above (resp. below) the diagonal line means that the solution found by ACO▷TS is worse (resp. better) than the one obtained by RPOLY.

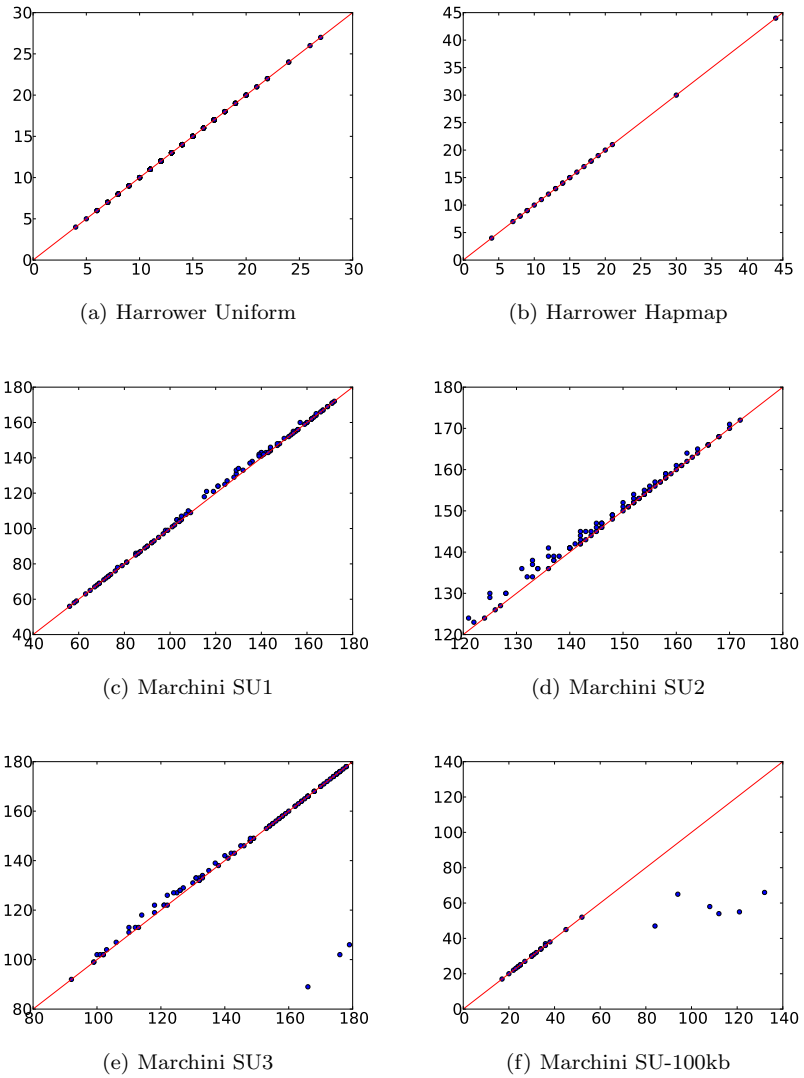
These results show that RPOLY and ACO▷TS are equally capable of solving to optimality the instances in Harrower Uniform, Harrower Hapmap, `Extra`, `ABCD`, `IBD` and `Climer` datasets. Only in `ACE` dataset ACO▷TS cannot solve two instances to optimality. The results show that RPOLY is also very effective on bigger datasets, namely Marchini's. On Marchini SU1 and Marchini SU3, ACO▷TS is almost as good as RPOLY, while in Marchini SU2 RPOLY seems slightly superior. On the other hand, ACO▷TS is clearly superior to RPOLY in Marchini SU-100kb as far as solution value is concerned since the ACO▷TS can either find the optimum whenever RPOLY does, or a better solution if RPOLY does not. It is interesting to notice that in six Marchini SU-100kb instances and three Marchini SU3 instances RPOLY yields a far worse solution than ACO▷TS because it cannot reach the optimum.[4] Likewise, on large-size instances such as `Eronen` and `Daly`, the ACO▷TS algorithm is particularly effective as it always dominates RPOLY.

We can conclude that RPOLY is adequate for small-to-moderate-size instances, while the ACO▷TS is comparable to RPOLY on the easiest datasets and seems the only viable option for large size problems.

3.4 Instance structure analysis

The two solvers RPOLY and ACO▷TS clearly exhibit a qualitatively different behavior on different instances also in the same instance set, therefore it is very

[4]We recall that we run the *anytime* version of RPOLY, so the optimality is not guaranteed.



(a) Harrower Uniform

(b) Harrower Hapmap

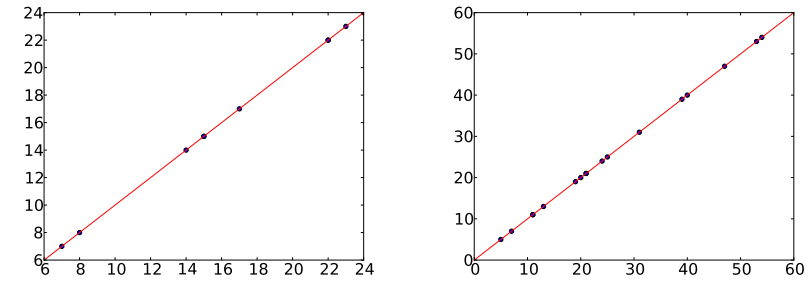
(c) Marchini SU1

(d) Marchini SU2

(e) Marchini SU3

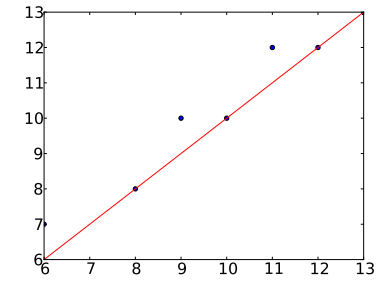
(f) Marchini SU-100kb

Figure 3.5: Comparison of solution value RPOLY (x-axis) vs. ACO▷TS (y-axis) on Harrower and Marchini datasets.

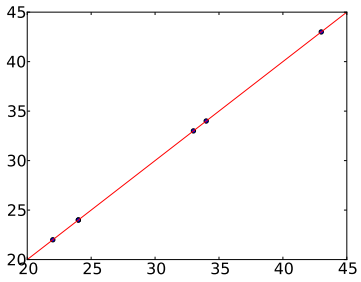


(a) ABCD

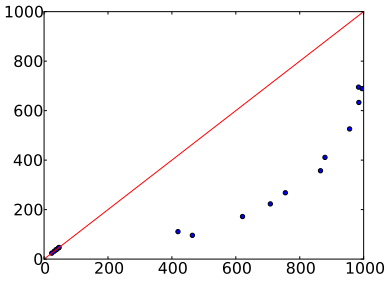
(b) IBD



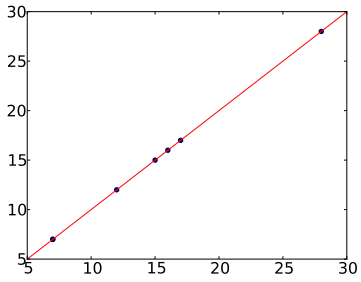
(c) ACE



(d) Extra

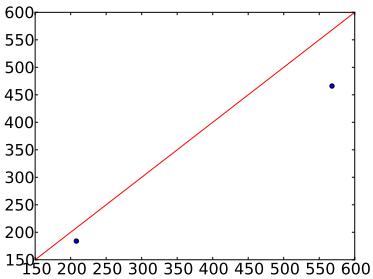


(e) Eronen



(f) Climer

Figure 3.6: Comparison of solution value RPOLY (x-axis) vs. ACO▷TS (y-axis) on datasets from [56, 73, 94].



(a) `Daly`

Figure 3.7: Comparison of solution value RPOLY (x-axis) vs. ACO▷TS (y-axis) on `Daly` benchmark.

important to study what features are most responsible for instance hardness. In our analysis, we partitioned the whole set of instances in three classes—*ACObt*, *ACOeq* and *ACOwt*—composed of the instances for which ACO▷TS returns a solution better than, equal to or worse than that returned by RPOLY, respectively.

For a Haplotype Inference instance several features can be considered, such as the number of genotypes, sites, heterozygous sites and also characteristics of the compatibility graph. We selected the following 16 features:

- *number* of genotypes;
- *number* of sites;
- statistics on the number of heterozygous sites: *min*, *max*, *1st* and *3rd quartile*, *mean* and *median* (6 features);
- *number* of edges in the compatibility graph;
- statistics on the compatibility graph vertex degree: *min*, *max*, *1st* and *3rd quartile*, *mean* and *median* (6 features);
- *Fiedler value* (λ_2) of the compatibility graph. λ_2 is the second smallest nonzero eigenvalue of the graph Laplacian matrix and provides information on the size of any graph cut [79].

Our conjecture is that the origins of the performance difference between ACO▷TS and RPOLY are very likely to be found where the combinatorial nature of the problem arises, that is, in some features related with the connectivity of the compatibility graph, such as the number of edges or the vertex degree. Boxplots in Figure 3.8 show a compact view of the distributions of the main features listed above.

From the boxplots, we can observe a common pattern: the sets *ACObt* and *ACOwt* can be separated—with some exceptions—by looking at graph features,

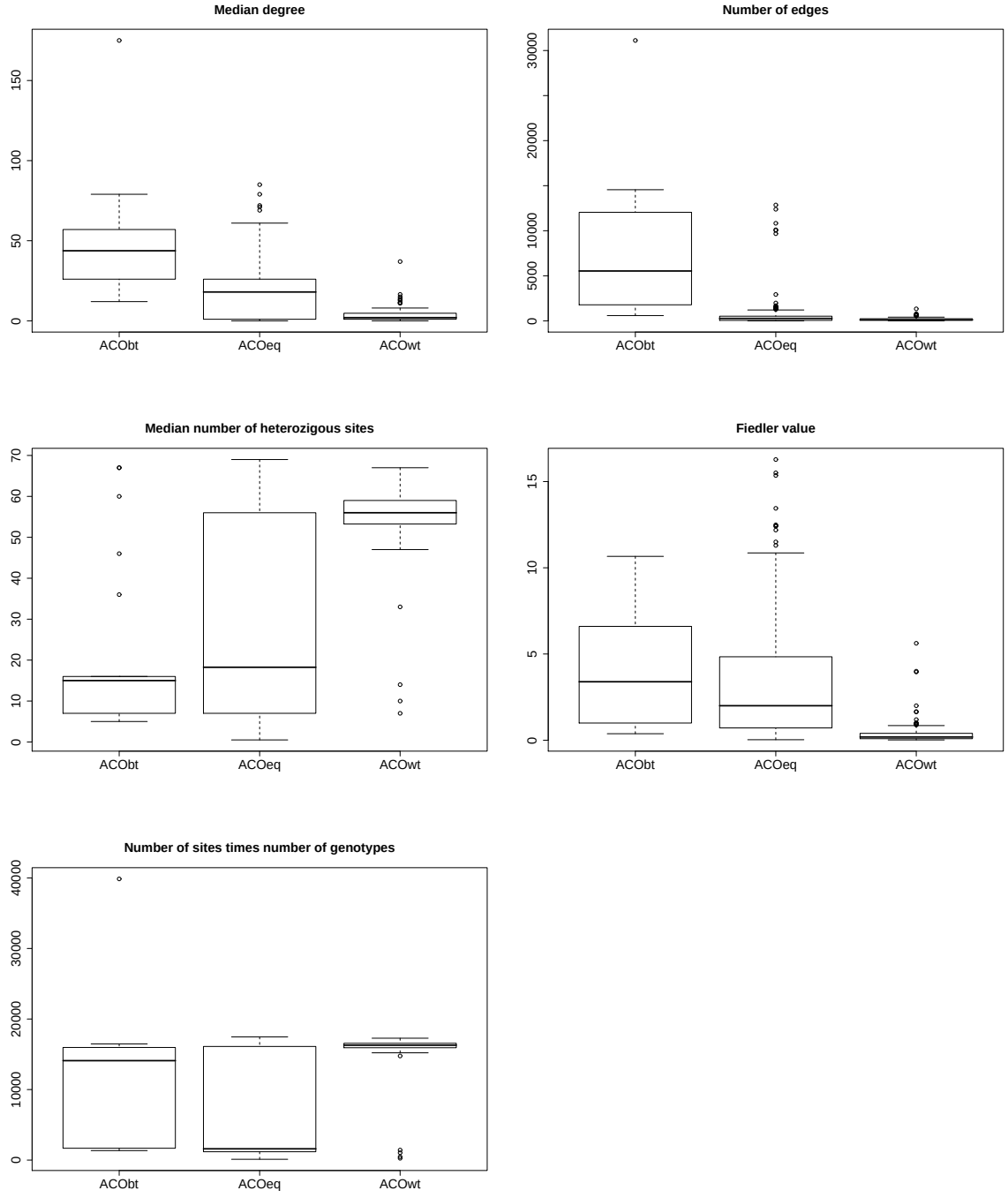


Figure 3.8: Boxplots of the distributions of the most significant features for the instances in the sets *ACObt*, *ACOeq* and *ACOwt*.

while the instance size (computed as the number of genotypes times the number of sites) does not discriminate very well—though, by looking separately at number of genotypes and number of sites the distinction is less unclear. Conversely, the distribution of the features in the instances composing the set *ACOeq* seems to be highly spread along all the range, therefore it is not possible to clearly discriminate among the three classes on the basis of a single feature. Nevertheless, the sharp separation between *ACObt* and *ACOwt* makes it possible to extract some interesting pieces of information about the relation between instance structure and algorithm performance. The most notable difference between the sets *ACObt* and *ACOwt* is that the compatibility graph associated to instances in *ACObt* is much more dense than for *ACOwt*, because the degree, the number of edges and the Fiedler value are much higher in *ACObt* than in *ACOwt*. This could probably be explained by the fact that the SAT instance generated for RPOLY increases with the number of compatibility relations between genotypes. Moreover, the median number of heterozygous sites is much lower in *ACObt* than in *ACOwt*: in this case, the lower the number of heterozygous sites per genotypes, the better ACO▷TS performs. An explanation for this phenomenon could be that the ACO part of ACO▷TS might not explore effectively the space of possible assignments for each genotype.

A more detailed analysis on the relation between compatibility graph features and algorithm performance should be undertaken in order to clearly identify the most discriminant feature. Furthermore, it would be of practical relevance to be able to choose the most appropriate solver, between RPOLY and our hybrid metaheuristic, on the basis of instance features. In this way, a quick test on some significant instance features can be used to select the solver that is more likely to perform better on that particular instance. This idea has been already applied in the context of combinatorial optimization problems, also exploiting machine learning techniques [42, 99, 141]. For example, it is possible to employ a machine learning technique to infer a decision tree, i.e., a set of rules or a more advanced classification method such as *AdaBoost* [82], to have a prediction of the class the instance will belong to.

3.5 Conclusions and discussion

We have shown the effectiveness of the approach of hybridizing an Ant Colony Optimization and a Stochastic Local Search metaheuristic for solving the Haplotype Inference problem under the pure parsimony criterion. The hybrid ACO▷TS algorithm improves the performance of its basic components and is very effective, especially on real-world large-size instances coming from the HapMap project. Moreover, the hybrid algorithm is able to compare favorably against the state-of-the-art solver for this problem.

Unfortunately, this approach cannot be extended to tackle instances with unknown without a major reformulation of the algorithm. As written in Section 2.3, the presence of unknowns radically changes the complementarity property, which is the basis of Clark's Rule and, hence, of the ACO algorithm. A possible solution to the problem of unknowns that still uses familiar, but different, notions, such as compatibility and complementarity, is the topic of Chapter 5. One could of course try to apply these algorithms to an instance with unknowns by substituting every missing site with a heterozygous site. This

operation, though, adds constraints to the problem and the solution returned by an algorithm, although feasible, will not likely be optimal.

For what the ACO component is concerned, this algorithm can be also seamlessly applied to the minimum entropy formulation; it suffices only to change the objective function, and consequently the quality function. It has to be noticed that the behaviour of lower level resolution (Section 3.2.1) is naturally geared towards parsimony so the performances of the algorithm might not be as good. Of course, from an engineering point of view, this algorithm is also modular in nature, so it is possible to replace the current lower level resolution with a procedure more suited to minimum entropy.

Chapter 4

Genetic Master-Slave Algorithm for Haplotype Inference by Parsimony

In this chapter, we introduce a general framework of a hybrid metaheuristic technique called "Master-Slave Genetic" algorithm (MSG). We will show how this framework can be successfully applied to a wide variety of combinatorial optimization problems for which a parametrized constructive procedure is available. We will see how the MSG can be used to enhance a constructive heuristic, similarly to what we did in Chapter 3 with the ACO algorithm, and we will describe its application to the Haplotype Inference by Pure Parsimony. In addition, with the aim of showing the generality of this approach, we also present the results of the application of the MSG to two graph-related problems, namely, the Capacitated Vehicle Routing Problem (CVRP) [55] and the Capacitated Minimum Spanning Tree Problem (CMSTP) [75].

Our formulation of MSG has been introduced in [20, 21] along with an application to Haplotype Inference. CVRP and CMSTP have been object of study in [14], where an effective MSG algorithm is proposed.

4.1 Introduction and motivations

In what follows, we present a "Master-Slave Genetic" algorithm (MSG) able to improve the performance of existing constructive heuristics in a more general setting. The MSG can be seen as a "two-level genetic algorithm‘" (TLG), where TLGs represent a broad category of metaheuristics sharing a common design principle. In a MSG algorithm, the problem resolution is decomposed into two interacting stages, in which the first stage consists of a genetic algorithm.

TLG algorithms are widely adopted and their paradigm is interpreted in different ways. For example, in [49] a TLG to solve a site allocation problem is proposed, in which a genetic algorithm defines first the location decisions, then an LP solver provides the corresponding flow assignment. A similar idea is employed for an allocation problem in [70], and in a flow interception problem [245]. TLGs are used in a different spirit in [87] for the Unit Commitment

problem. A GA generates a solution (chromosome) for a relaxed problem, which is then repaired by a second algorithm. A TLG for the Joint Product Family Platform Selection and Design problem is has been also presented [130].

On the other hand, several works [136, 192, 207], interpret the TLG paradigm in the same way as in our contribution. The solution construction is decomposed into two stages: a high-level GA produces first a permutation of elements in a set (chromosome), then, given the chromosome information, a lower-level solver computes and evaluates the solution value. This value is then returned to the GA, so as to perform selection.

Even if our method is similar to the ones presented in the previous cited works, there is one major innovation: instead of devising an *ad-hoc* lower-level heuristic, we use and enhance an existing constructive procedure. For Haplotype Inference, the proposed algorithm is once again based on a deterministic variant of Clark's Rule whose parameter is the genotype ordering. The algorithms for the graph problems are instead built upon the concept of *saving heuristics*, deterministic constructive algorithms whose behaviour depend on an input list of merge operations which, essentially, determines the final feasible solution.

The remainder of this thesis is structured as follows. Section 4.2 is devoted to the description of the MSG; Section 4.3 describe our MSG algorithm for Haplotype Inference by Pure Parsimony; Section 4.4 reports the results about the experiments on the haplotyping problem. Since this dissertation is mainly focused on biological applications, the discussion about the two graph problems is concern of the last two Sections A.1 and A.2, whose scope is limited to a brief introduction to the new problems and a summary of the experiments. Further details and results are available in a separate work [14].

4.2 Master-slave Genetic framework

This section introduces the MSG framework in general terms.

Algorithm 3 Master-slave high-level framework

1: $\mathcal{P} \leftarrow$ buildInitialPopulation(n) {set of n individuals}
2: evaluate($\mathcal{P}$)
3: **while** terminating conditions not met **do**
4: $\quad\mathcal{P}' \leftarrow$ applyGeneticOperators($\mathcal{P}$) {operators depend on individual representation}
5: $\quad$evaluate($\mathcal{P}'$) {use slave algorithm}
6: $\quad\mathcal{P} \leftarrow$ populationUpdate(n, $\mathcal{P}$, $\mathcal{P}'$)
7: **end while**
8: **return** min($\mathcal{P}$)

The core idea of our MSG lays in the possibility of splitting the solution construction in two nested phases. In the first phase, the parameters of a solution construction procedure are set by a *master* solver and in the second phase the solution is actually built by a *slave* solver. For example, for combinatorial problems there can be constructive procedures based on: the sequence of objects to be included in the solution and/or the decisions to be taken, the set of preassigned variables or the set of hard constraints to be fulfilled.

In a sense, the problem is decomposed into two, interdependent, sub-problems. The solution to the first problem is an input for the second, that actually constructs a solution. In our TLG framework, the first algorithm is a genetic algorithm, while the second is a problem-specific heuristic. We can say that, in a way, the master explores a search space of "parameters": the objective value of the points in this space is the value of the solution returned by the slave.

From an implementation point of view two things can be further noted. From an algorithm design perspective, the main advantage of partitioning a problem into master and slave is a clearer separation of concerns, which helps design a more extensible algorithm and allows a simple and neat implementation. In addition to that, an MSG algorithm lends itself to parallel implementations as it is, at its core, a genetic algorithm. The slave algorithm could be executed on different processors, up to one for each offspring to be evaluated. This would save up most of the computing time, because the evaluation of the fitness is the most time consuming task in MSG.

After having fleshed out the its general characteristics, we see that, in order to use the MSG framework, the problem-specific design choices amount to deciding a constructive for the slave and the master's relevant components (genetic operators, update strategy and so on).

Software Frameworks for MSG. From a programmer's perspective, a MSG algorithm can be easily implemented on top of an existing genetic algorithm framework library in a straightforward way, as was pointed out at the end of the previous section. Every software library that facilitates genetic algorithm development provides, in fact, some way to define a problem-specific fitness function; that hook can be utilised to implement the desired slave procedure.

Though many software frameworks are available [1], we felt the need to have a tool better suited to our needs. As part of the work presented in the following sections, we implemented ourselves a `C++` software framework, called `EasyGenetic` [17], to ease the development of MSG algorithms. The main difference between `EasyGenetic` and other popular evolutionary algorithm frameworks, such as ParadisEO [41], `galib` [231] or ECJ [68], is that it is heavily based on template metaprogramming techniques [2] to eliminate runtime abstraction penalties and to allow more flexibility in design.

Many GA frameworks provide typical chromosome representations, typically one- or two-dimensional numeric arrays; other frameworks also include tree-based chromosomes for genetic programming. In `EasyGenetic` we take a different standpoint. Inspired by another software framework, `spGAL` [215], `EasyGenetic` is designed to encourage the programmer to write reusable generic code. "Generic" in this context means that every component abstracts away from the actual implementation of the chromosome. Chromosomes can thus be as simple as bitstrings or as complex as trees, graphs or totally different objects altogether. The only constraints between algorithm components (recombination operators, selection strategy and so on) and chromosomes is represented by *concepts*. Concepts are specifications of structural interfaces, as opposed to the usual definition of interface found in mainstream OOP languages, such

[1] At the time of the writing, there are 284 projects on SourceForge [214] categorised under "Genetic Algorithm", 88 of which are written in `C++`, our language of choice.

[2] ParadisEO makes use of design patterns based on *template instantiation*, but it does not exploit metaprogramming techniques.

as Java, and are of fundamental importance in writing generic functions [98]. Concepts are a way to specify structural subtyping, that is, a form of subtyping that is roughly based on which operations a type provides. As opposed to nominal subtyping, where the programmer must explicitly state the subtype relation, concepts allow for a more "open" form of subtyping: every type that meet a concept's requirements can be automatically used, without programmer intervention, wherever an object of that particular concept is required. For instance, a sorting function might require that the objects it sorts all implement a comparison operator. A similar form of subtyping, popular among dynamical languages, is called Duck typing [67], with the main difference that structural subtyping with concepts is enforced at compile-time, while Duck typing is checked at run-time. With concepts and metaprogramming, we are able to have a great deal of expressiveness without sacrificing compile-time checks and runtime efficiency. For example, `EasyGenetic` can support crossover operators that take an arbitrary number of parent chromosomes, say n, and, contemporarily, are naturally implemented by a simple functions with n arguments. Other frameworks, instead, allow only a fixed number of recombinants, typically one (asexual crossover) or two (sexual crossover) [231].

Further information on `EasyGenetic` is available in [21, 22].

4.3 MSG for Haplotype Inference

Like we previously did in Chapter 3, we enhance a parametrized constructive by means of a metaheuristic. Yet again, our choice falls on Clark's Rule. In this section we first describe a deterministic variation to Clark's Rule (Section 4.3.1), which will take the role of our slave procedure, and then we detail the implementation of the master component (Section 4.3.2), a steady-state genetic algorithm, by describing its characteristics.

As usual, a preprocessing step is applied beforehand and the input instance is partitioned into independent sub-instances (see Section 2.2.2).

4.3.1 Slave algorithm

Here we describe the Clark's Rule-based slave algorithm. The constructive is a deterministic procedure, similar to the lower level ACO algorithm presented in Section 3.2.1, which depends solely on the genotype visiting order (formally a permutation) supplied by the master. The choice of a deterministic slave procedure has one major advantage over a stochastic one because the evaluation of the solution returned by the master, i.e., a permutation of genotypes, has a unique evaluation. In case of a stochastic algorithm for the slave, solution quality would be a stochastic variable and one would need to estimate it, for example by taking the average of a sample population. This issue can be tackled by using techniques adopted for algorithms tackling stochastic problems [28].

The algorithm starts from the first genotype of the permutation π supplied by the master. At each step, the visited genotype is removed from π. The procedure then deterministically chooses a number of haplotypes to be included in the partial solution H. During haplotype selection the three possible scenarios that could arise are the ones already mentioned in Section 3.2.1. Save for the trivial case in which H already contains a pair of haplotypes resolving

the current genotype g, there are three different possibilities: *(i)* no resolving candidates in H_ϕ; *(ii)* one candidate; and *(iii)* more than one candidate. In each case, haplotype selection works as follows:

Case (i): Scanning π from the left, the first non-visited genotype $g' \sim g$ is taken and, deterministically, a haplotype h that solves both g' and g is computed. If no such g' exists, h is equal to the string g with its ambiguous sites set to 0. h and $g \ominus h$ are added to the partial solution.

Case (ii): When only one resolvent candidate h is available in H, $g \ominus h$ is added to the partial solution. This is a straightforward application of complementarity.

Case (iii): When more than one candidate is present in H, the procedure chooses deterministically from these using a heuristic similar to case *(i)*. Let H' be this set of candidates: the procedure searches in π for the first unvisited genotype $g' \sim g$, and chooses first haplotype in lexicographical order from H' that explains both. In case no such haplotype exist, the next compatible genotype in the permutation is tried. If the whole permutation is scanned and no haplotype was selected, the choice of whatever haplotype in H' would not affect the future behaviour of the heuristic [3] and the first haplotype in lexicographical order is thus chosen.

The algorithm stops when all genotypes have been visited.

In order to deterministically build a haplotype that resolves two compatible genotypes g and g', we can use the following method (compare with Section 3.2.1 where lower-level genotype resolution is guided by pheromone values). First the conflation c of g and g' is calculated according to these rules:

$$g_p = 0 \vee g'_p = 0 \quad \Rightarrow \quad c_p = 0 \tag{4.1a}$$
$$g_p = 1 \vee g'_p = 1 \quad \Rightarrow \quad c_p = 1 \tag{4.1b}$$
$$g_p = 2 \wedge g'_p = 2 \quad \Rightarrow \quad c_p = 2 \tag{4.1c}$$

for all $p = 1, \ldots, m$, where m is the genotype length. All remaining heterozygous sites in c are subsequently set to 0.

4.3.2 Master algorithm

As we wrote previously, in order to use the constructive heuristic *à la* Clark in Section 4.3.1, we have to provide an effective genotype resolution order. Learning such ordering is, ultimately, the objective of the master. The proposed version of the MSG uses a master genetic algorithm to compute a permutation of genotypes.

To fully define a genetic algorithm, one has to specify:

- the chromosome structure;
- a selection procedure;

[3] In such a case all haplotypes in H would be incompatible with all unvisited genotypes.

- a population update procedure;
- genetic operators and how they are applied to the population;
- an evaluation criterion for assigning fitness to the individuals;
- an initialization step to generate an initial population.

In the rest of this chapter we will call "individuals" or "chromosomes" the entities manipulated by the genetic algorithm so as to avoid possible confusion with the genotype strings which make up a Haplotype Inference instance.

An individual represents a single permutation of genotypes, that is, a genotype ordering. That ordering is crucial for the slave procedure to compute a good solution. The initial population is randomly generated. The evaluation criterion is provided by the slave algorithm, that builds a solution and computes its cardinality. The fitness function, higher for fitter individuals, is equal to the quality function defined in Section 3.2.1 for the ACO. The selection procedure is a standard roulette-wheel, in which individuals are taken for mating with a probability proportional to their respective fitness values. As for population update, we chose to implement a steady-state genetic algorithm. Concerning genetic operators, we adopted the usual definition of crossover and mutation for permutations of length n, which are detailed in the following.

Mutation: the mutation of an individual encoded as a permutation simply involves a random swap of two elements. The average number of swaps that can be performed on each chromosome is an algorithm parameter, called *mutation rate* or m_r for short.

Crossover: first a random point-cut is chosen and the permutations p, q are split into two sub-sequences $(p_1, p_2), (q_1, q_2)$. Then the new permutations p' and q' are constructed in this way[4]: at first $p' = p_1$, then all the elements of q_2 which are not in p_1 are orderly appended to p'; if the length of p' is still less than n, the procedure keeps appending elements to p' taken from sub-sequence q_1 if the given elements are not already present in p_1. Crossover operator is always applied.

4.4 Experimental Analysis

To investigate the performance variance with respect to parameter values and to measure the contribution of the learning component of the genetic algorithm, we have run four different versions of the algorithm:

***ga*:** the plain master-slave genetic algorithm;

***ga*$^+$:** the master-slave genetic with increased computational resources;

***ga*$_{no}$:** the *ga* version of the algorithm, but with learning component disabled;

***ga*$\triangleright$*TS*:** is the *ga* hybridized with the local search procedure introduced in Section 3.2.2 (see below).

[4] We take into account only the construction of p', since the other is symmetrical.

The parameters of *ga* have been tuned by means of a full-factorial analysis on a restricted number of instances and the best candidate according to the the average solution quality has been selected. To estimate the improvements achievable with a larger amount of computational resources, ga^+ was also tested: this version runs with twice the population size and idle iterations with respect to *ga*. The impact of the GA learning mechanism on search has also been evaluated by running *ga* with the selection operator disabled ($m_r = 0$ and no crossover); the resulting algorithm, ga_{no}, thus performs a large number of independent runs of the constructive procedure. Finally, we combined the master-slave constructive procedure with a local search post-optimization. As explained previously in Section 3.2.3, for $ga \triangleright TS$ we chose to have a simple integration to better asses the contribution to solution quality of each algorithm. For this reason, the two algorithms have been serialized: first the master-slave is run and the best solution returned is passed to the local search as initial state. In Table 4.1 the main parameters of the algorithms are summarized. Each generation offspring size is equal to population size. In case of $ga \triangleright TS$, maximum iterations and maximum idle iterations parameter are the same for both the GA and the local search.

Variant	m_r	**pop. size**	**max iter.**	**max idle iter.**
ga	1	100	500	100
ga^+	1	200	500	200
ga_{no}	0	100	500	100
$ga \triangleright TS$	1	100	500	100

Table 4.1: Parameters of the algorithms.

We will first present the comparison among the variants of the master-slave solver. Then, in Subsection 4.4.1, we will show the results of the comparison of the master-slave solver against our ACO introduced in Chapter 3.

The algorithms have been tested on the same well known benchmarks for Haplotype Inference used previously in Section 3.3.1, namely, Harrower Hapmap, Harrower Uniform, Marchini SU1, Marchini SU2, Marchini SU3 and Marchini SU-100kb (their main characteristics can be looked up in Table 3.1). Furthermore, in a second experimental phase, we also performed experiments on new test sets, in order to measure the scalability of this new MSG algorithm with respect to the ACO hybrid. This new benchmark is composed by three test sets, namely HapMap CEU, HapMap YRI and HapMap JPT+CHB, which contain large-size instances extracted from real HapMap data. [5] Their characteristics are also summarized in Table 4.2. Although these benchmarks are not biologically plausible, they proved to be particularly difficult and are therefore suited to the objective of this experimental stage. This second experimental phase corresponds indeed to a validation of previous results, because algorithms were not tuned for these new instance sets [27].

The algorithms in the MSG framework have been developed in `C++`; the software was compiled with the `GNU` g++ compiler v. 4.1.3 (`-O3` optimisation enabled) and tested on a Intel Pentium 4 3.0GHz machine running Ubuntu 7.10

[5] Datasets are downloadable from `http://apice.unibo.it/xwiki/bin/view/HaplotypeInference/JHCR_2009/`.

Benchmark set	N. of inst.	N. of geno.	N. of sites
HapMap CEU	25	90	200
HapMap YRI	25	90	200
HapMap JPT+CHB	24	90	200

Table 4.2: Main features of the new hard benchmarks.

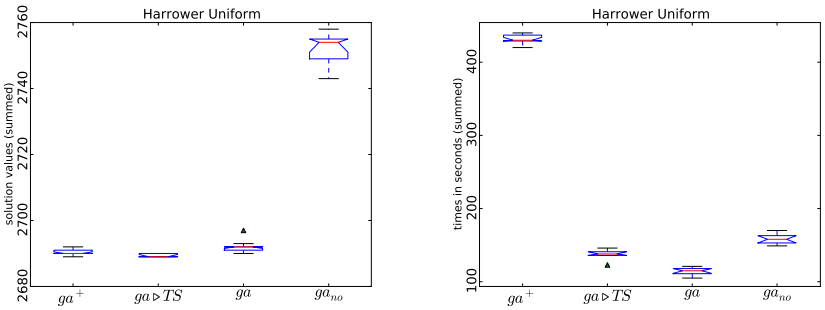


Figure 4.1: Harrower Uniform: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

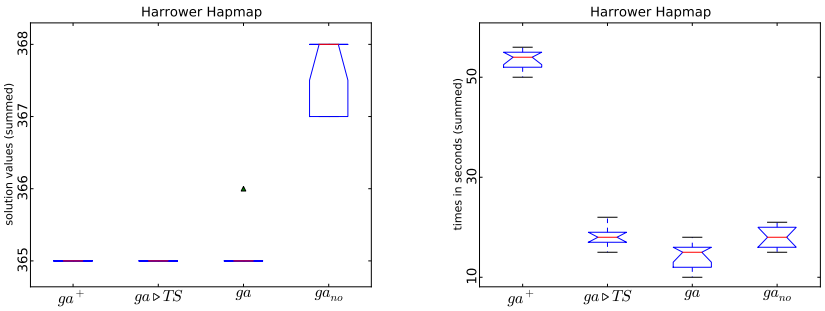


Figure 4.2: Harrower Hapmap: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

(kernel 2.6.22).

The aim of the experimental analysis is to compare the overall performance of the algorithms in terms of solution quality and execution time on each instance set. This analysis should answer two main questions: *(a)* does the genetic machinery, i.e., the learning mechanism of GA, improve the heuristic construction? And *(b)* is there any advantage in terms of solution quality or time in using this master-slave approach for Haplotype Inference?

The experimental analysis is composed of two parts: in the first we compared solution quality and running time on each complete instance set. The result of this analysis makes it possible to compare the global performance of each algorithm and then observe advantages and disadvantages of them. Results of

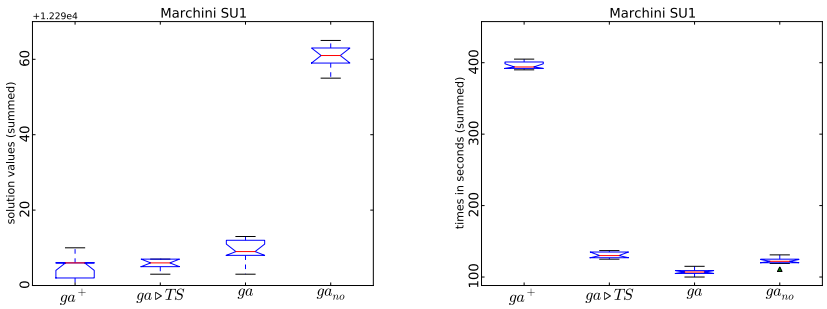


Figure 4.3: Marchini SU1: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

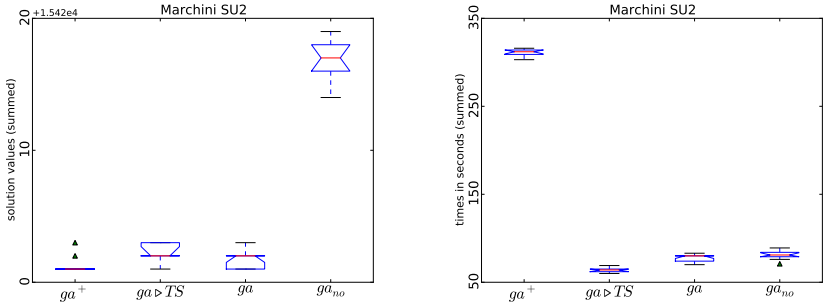


Figure 4.4: Marchini SU2: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

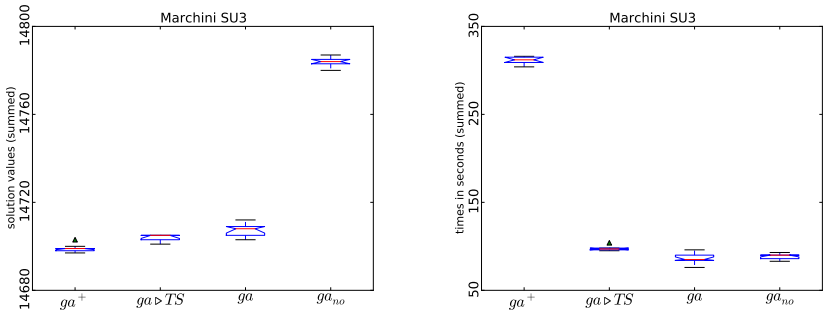


Figure 4.5: Marchini SU3: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

this comparisons are shown with boxplots. We also compared the algorithms pairwise and applied a Wilcoxon paired test [57] for assessing the superiority of one algorithm over the other in terms of solution quality. This test is applied on each instance set and compares algorithms instance-wise. The results of this

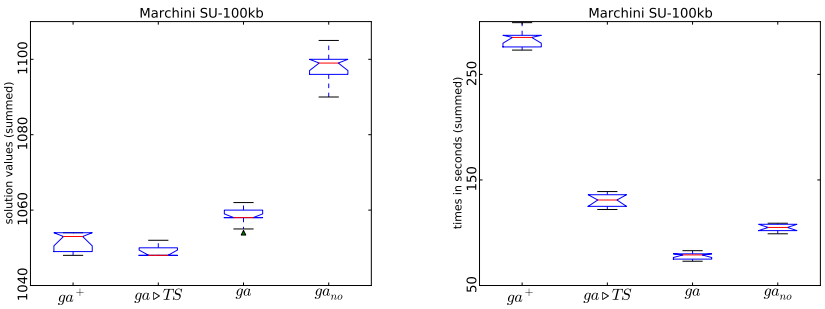


Figure 4.6: Marchini SU-100kb: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

analysis confirm the ones of the boxplot analysis.

In Figures 4.1–4.6, boxplots relative to solution quality (left) and running times (right) are drawn. The plots represent statistics of ten independent runs of the algorithms on all the instances of each set. Since our goal is to evaluate algorithm performance on each entire set of instance—and instances of each set are homogeneous—values shown are sums of solution values and running time on all the instances.

Boxplots are drawn with *notches*, which make it possible to assert when the medians of any two distributions are statistically different: if the notches of two boxplots do not overlap then there is statistical evidence that the two medians differ [47].

If we consider ga and ga_{no} graphics show that the learning component has a remarkable positive impact on solution quality while the cost in terms of computational resources is negligible. This answers question *(a)* and proves that a simple constructive procedure can benefit from a careful selection of genotype resolution order.

In general, all solvers (except for ga_{no}) have the desirable property of being stable: although they are stochastic algorithms, the distribution of the solution values is rather packed around the mean and even zero for Harrower Hapmap.

As for solvers ga^+ and $ga \triangleright TS$ their results in term of solution quality are superior to ga, even though the improvement is rather limited and, for what ga^+ is concerned, its execution times is considerably higher than that of ga. On the contrary, $ga \triangleright TS$ is not as slower than gaas ga^+, meaning that the contribution of Tabu Search to the runtime figures is limited.

Results on the new three instance sets, whose boxplots are depicted in Figures 4.7–4.9, are particularly meaningful as they confirm the same relation among the algorithms on a (hard) validation set. Moreover, it is worthwhile to note that these results emphasize the effectiveness of the learning component that considerably leverages the performance of the constructive heuristic, thus providing an answer to question *(b)*.

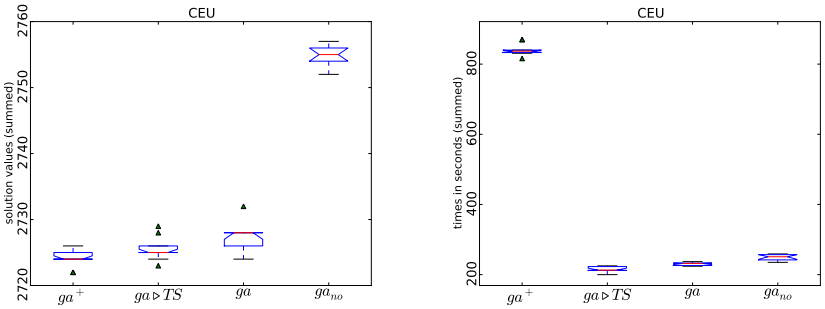


Figure 4.7: HapMap CEU: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

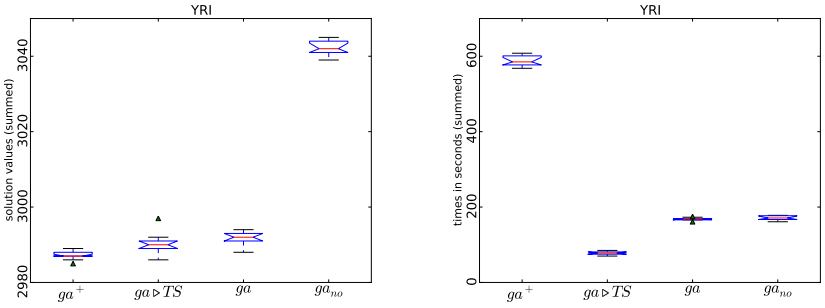


Figure 4.8: HapMap YRI: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

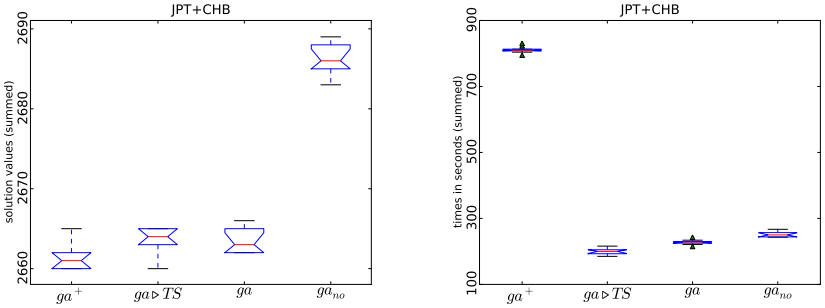


Figure 4.9: HapMap JPT+CHB: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

4.4.1 Comparison with the Hybrid ACO

To assess the overall performance of the master-slave algorithm, we compared ga, $ga^+$and $ga \triangleright TS$ with the hybrid solver described in Chapter 3, namely ACO$\triangleright$TS.

ACO▷TS was run with the same parameter settings reported in Section 3.3 and a limit on both total iterations and idle iterations of, respectively, 500 and 100. In Figures 4.10–4.18 the boxplots represent the statistics of the four algorithms w.r.t. solution quality and running time. We observe that the overall solution quality returned by our master-solver is slightly lower than that of the hybrid solver, but the execution time is remarkably shorter meaning that our MSG is solid choice for a fast upper bound procedure.

In order to show the effectiveness and generality of our MSG, we also present the results attained by the technique applied to two hard combinatorial graph-related problems, namely, the Capacitated Vehicle Routing Problem (Section A.1) and the Capacitated Minimum Spanning Tree (Section A.2). Material in both sections is taken from [14]. Since these applications are not related to genomics or biology in general, their discussion is delayed to Chapter A in the appendix.

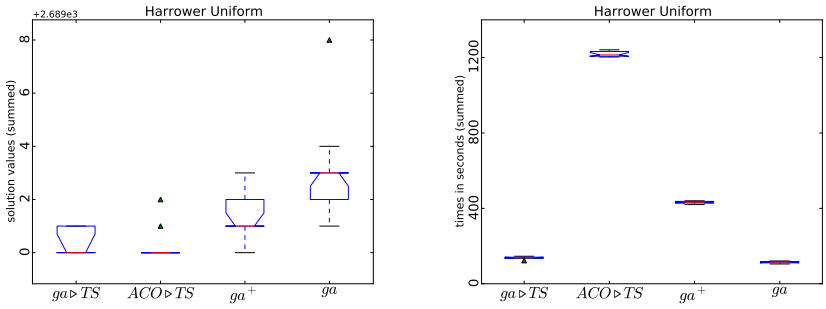


Figure 4.10: Harrower Uniform: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

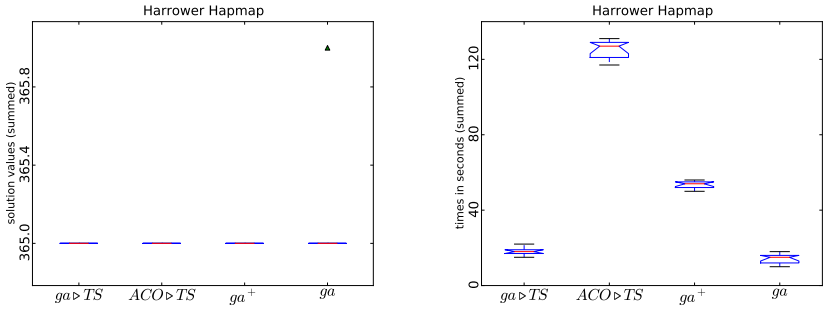


Figure 4.11: Harrower Hapmap: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

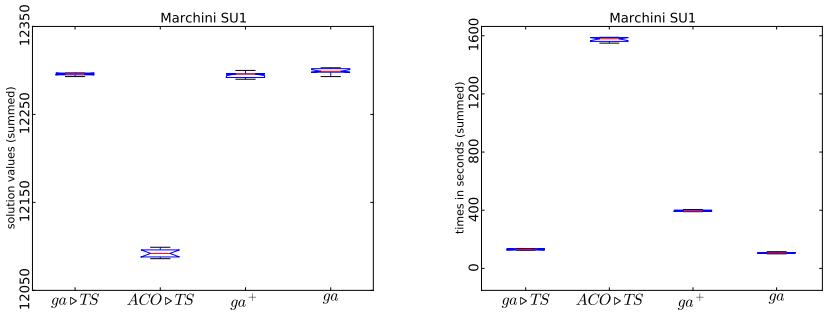


Figure 4.12: Marchini SU1: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

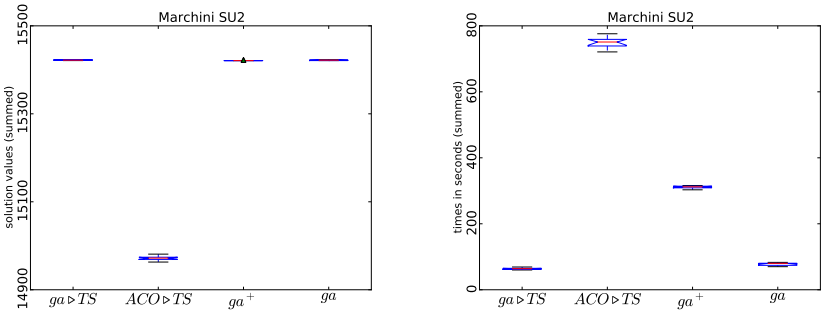


Figure 4.13: Marchini SU2: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

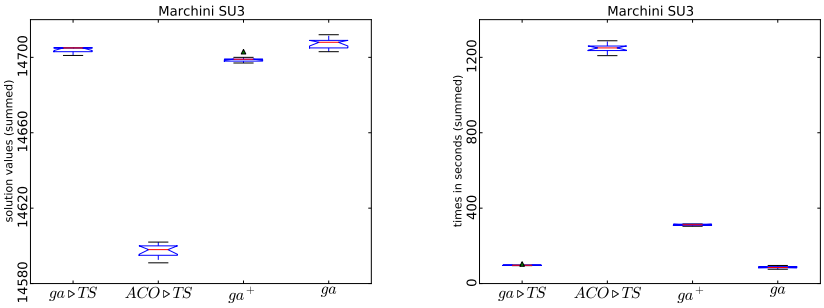


Figure 4.14: Marchini SU3: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

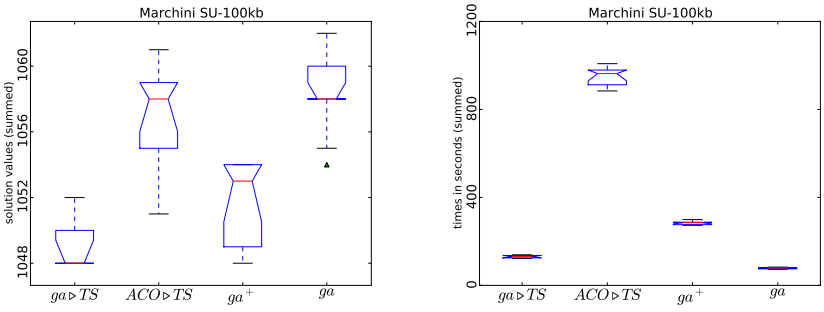


Figure 4.15: Marchini SU-100kb: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

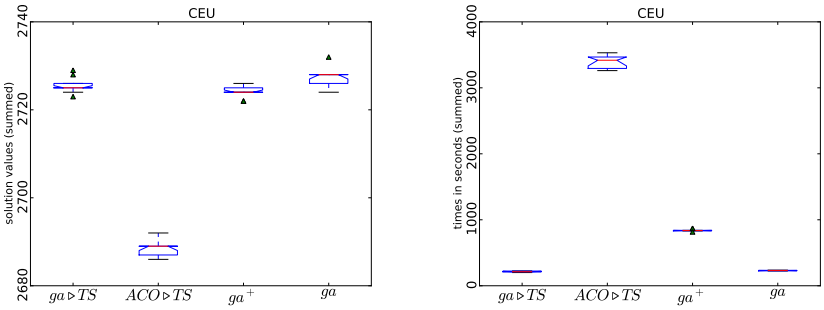


Figure 4.16: HapMap CEU: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

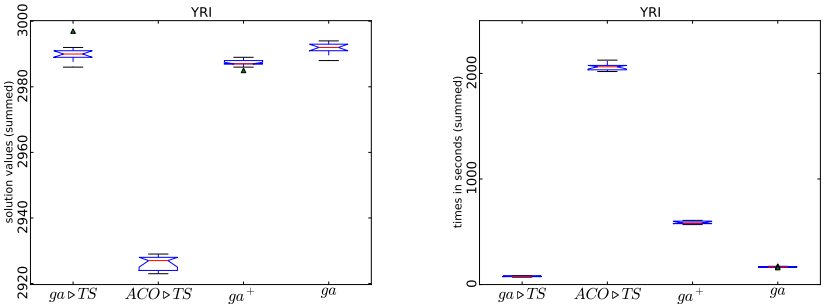


Figure 4.17: HapMap YRI: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

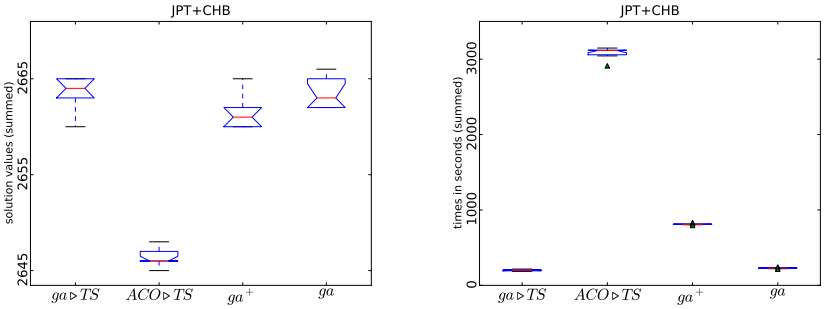


Figure 4.18: HapMap JPT+CHB: algorithms comparison in terms of solution quality (left) and running time (right) summed-up over all the set instances.

Chapter 5

Alternative Approaches to the Haplotype Inference Problem

In this chapter we introduce two original contribution to the Haplotype Inference.

In Section 5.1 we try to address one drawback of the metaheuristic approaches presented in previous chapters and will discuss a more general resolution framework for haplotype inference, which is based on an extended version of the compatibility rule. This approach, which makes use of some constraint programming (CP) techniques, is particularly suited to solving instances with unknown sites. The possibility of integrating CP and metaheuristics has already been fruitfully explored in the literature [155, 156]; for this reason we think that this approach might be attractive over other non-CP approach.

Section 5.2 is devoted to the description of an algorithm that aims to return a set of haplotypes whose elements are compatible to as many genotypes as possible. This haplotype set could indeed represent an important piece of heuristic information in constructive procedures that tackle not only Maximum Parsimony, but also Minimum Entropy. This algorithm is based on solving the Maximum Clique problem on the compatibility graph of an instance. We also recall that a related problem, the Maximum Independent Set, is the basis for a lower bound for the Haplotype Inference.

Both contributions have been implemented in working software prototypes. Although preliminary test have been conducted that confirm the correctness of the approaches, we are not going to present any experimental data. Further development is subject of ongoing work.

5.1 Enhancement of resolution by constraint programming techniques

The metaheuristic algorithm proposed in Chapters 3 and 4 both relied on Clark's Rule. The algorithmic framework makes it so that the experience accumulated during the search process is used to guide the resolution of a genotype by in-

cluding in the partial solution a haplotype likely to be used in the future to resolve other genotypes. This was made to address the ambiguity that would arise when applying Clark's Rule on a genotype that is compatible with more than one haplotype in the current partial solution. In such cases, bad decisions would cripple further search or would prevent the search to find a feasible solution. Exploitation of metaheuristic approaches indeed mitigates these problem, but still poor early decisions could severely impact the quality of the final solution by constraining later choices.

In this section, we extend the algorithmic framework we presented in previous chapters by borrowing some ideas from constraint programming techniques. We also introduce the concept of *generalised haplotype* and we show how this new conceptual framework can address some drawbacks of the discussed metaheuristics.

Problems of early commitment. Up to now, the algorithmic framework which underlies all metaheuristics for Haplotype Inference described in this dissertation can be summarised as follows: 1. Clark's Rule, or a variant thereof, is applied to each genotype, and 2. a high-level adaptive procedure selects each time which genotype to solve. The biggest problem with this approach is that the application of Clark's Rule, which is essentially the definition of complementarity, is too restrictive: every time the complementarity definition is applied, a phasing for a genotype is generated and that choice cannot ever be retracted. This approach places uselessly stringent constraints that limit future choices performed by the search procedure.

For instance, let us consider two genotypes $g_1 = 0120222$ and $g_2 = 0222012$; if we want to resolve g_1 a myopic constructive would generate a random pair of haplotypes, but we could miss the opportunity to add to the current solution a haplotype also compatible to g_2: haplotypes such as 0110110 or 0100001 are obviously bad choices. We could search, as shown in Section 3.2.1, for all haplotypes compatible to g_1 and g_2, like $h_1 = 0100011$ or $h_2 = 0110010$; but then we might have to solve a third genotype $g_3 = 0100012$ and only h_1 seems to be the most parsimonious choice because is compatible to g_1, g_2 and g_3 as well.

This small example demonstrate the danger of early commitment. What we would like to do is to have the possibility to include both h_1 and h_2 in the solution and then to settle for one of them only when the need arises (in the example, when we resolve g_3). We show how we can do it in a reliable way by introducing generalised haplotypes.

5.1.1 Haplotype Inference with generalised haplotypes

Informally, a generalised haplotype represents a set of (ordinary) haplotypes. More formally, a generalised haplotype is a vector containing zeros, ones or binary variables. A generalised haplotypes like $01xy0$ represents the haplotype set $\{01000, 01010, 01100, 01110\}$ obtained by instantiating x and y in all possible ways.

The compatibility relation is an extension of the one in Section 2.2.2. What we basically do is to make a variable compatible to any site, be it homozygous (0 or 1) or heterozygous (2); for instance, $01xy0$ is compatible to 02020 and

	0	1	2
0	0	—	1
1	—	1	0
x	0 and set $x = 0$	1 and set $x = 1$	y and post constraint $x \neq y$

Table 5.1: Rules to compute $g \ominus h$. The first row indicates the genotype value at the i-th site, the first column indicates the i-th site of h; x is a generic binary variable in h, while y generically refers to a newly created binary variable. Each table entry contains the value at site i of the new haplotype $g \ominus h$ (entries marked with '—' correspond to impossible cases). When variables are involved an action is also specified.

22010. This is a natural consequence of the fact that a generalised haplotype represents, in fact, a set of haplotypes.

The complementarity property, instead, is radically different. Basically, when we compute $g \ominus h$, where h is a generalised haplotype, we want to construct a generalised haplotype $\overline{h}$ such that for every possible complete assignment of all variables in h, $\overline{h}$ is such that $h \oplus \overline{h} = g$. By enforcing the rules summarised in Table 5.1 we attain this objective. For each position i in haplotype h we compute the corresponding element of $\overline{h}$ according to the rules in Table 5.1. When the i-th site in h is either 0 or 1, we have the same inference rule as for the ordinary complementary. When the i-th site is a variable, generically indicated with x in the table, we introduce a new variable, generically indicated with y in the table, used nowhere else in the algorithm and execute an action according to the rules stated in the bottom row of Table 5.1; such variable becomes the i-th element of $\overline{h}$.

In an eventual algorithm, these relations can be easily implemented in practice by constraint programming facilities. Notice also that, computationally speaking, the complement operation has now *side effects* on the state of the algorithm.

Let us review the previous example utilising generalised haplotypes. We solve $g_1 = 0120222$, $g_2 = 0222012$ and $g_3 = 0100012$ in this order. To solve g_1 we build haplotype $h_1 = 01x_10x_2x_3x_4$ and, by complementarity, we calculate $\overline{h_1} = g_1 \ominus h_1 = 01y_10y_2y_3y_4$ where $x_i \neq y_i, i = 1, 2, 3, 4$. Then we resolve g_2 with, say h_1 (the outcome would be identical if we used $\overline{h_1}$); the computation of $g_2 \ominus h_1$ yields $h_2 = 00z_1101z_2$ and, as side effect, posts constraint $z_1 \neq x_1 \wedge z_2 \neq x_4$ and sets $x_2 \leftarrow 0$, $x_3 \leftarrow 1$; by constraint propagation now $h_1 = 01x_1001x_4$ and $\overline{h_1} = 01y_1010y_4$. We can now use h_1 (or $\overline{h_1}$) to solve g_3: we compute $h_3 = g_3 \ominus h_1 = 010001w_1$, post $x_4 \neq w_1$ and set $x_1 \leftarrow 0$; after propagation, $h_1 = 010001x_4$, $h_2 = 001101z_2$, $\overline{h_1} = 011010y_4$. Notice that now it is impossible to obtain a solution with more than 4 haplotypes regardless of the genotype visiting order.

This example shows that if we employ generalised haplotypes, we have to take into account that every time a complement is calculated, variable values are possibly updated and new constraints might be posted. In the end, the solution returned is not just a set of ordinary haplotypes: it is, instead, a set of generalised haplotypes *and* the corresponding constraints among them; it might also contain unassigned variables. This is a major difference between this approach and one based on ordinary haplotypes: the solution returned by

	0	1	2	9
0	0	—	1	y
1	—	1	0	y
x	0 and set $x = 0$	1 and set $x = 1$	y and post constraint $x \neq y$	y

Table 5.2: Rules to compute $g \ominus h$ when g contains unknowns (9). Symbols and interpretation are explained in Table 5.1 and related paragraph.

an algorithm, which we can call *generalised phasing*, constitutes, in fact, a *set of phasing* obtained by instantiating unassigned variables in all possible ways. This can be practically very useful in algorithm integration. Different phasings could be produced according to diverse criteria like, for instance, the most likely with respect to a statistical model, or the optimal according to some secondary objective function, like Minimum Entropy.

It has not been demonstrated, though, that all phasings implicitly described by a generalised phasing are equivalent with respect to parsimony (i.e., contain the same number of haplotypes). This is particularly relevant in the event one wants to utilise generalised haplotypes in a multiobjective optimisation problem.

Generalised haplotypes and unknowns. An important limitation of algorithms based on the ordinary complementarity definition is their inability to effectively cope with unknowns. The reason is that when unknowns are involved haplotype complement does not yield a unique result anymore (see Section 2.3). If we introduce generalised haplotypes the problem is automatically solved: we only need to slightly change complementarity rules in Table 5.1. Specifically, we add a single column to take into account unknown sites. Table 5.2 shows the new definitions which can be summarised as follows: the complement of an haplotype site of any kind with respect an unknown site is a fresh variable.

Use case: Master-slave genetic with generalised haplotypes

.

Let us now reinterpret our master-slave algorithm from Section 4.3 using generalised haplotypes. The bulk of the algorithm is essentially the same: we still have a master genetic and a slave algorithms. The adoption of generalised haplotypes, fundamentally, changes the definition of complementarity, so the only real modification is limited to the slave (see Section 4.3.1) with enacts the genotype resolution. Let us review the three scenarios that the algorithm encounters when resolving a genotype, i.e., *(i)* no resolving candidates in the current partial solution; *(ii)* one candidate; and *(iii)* more than one candidate. Here the compatibility relation is the one for generalised haplotypes of course.

Case (i): A new generalised haplotype h is built: just substitute every 2 in g with a fresh binary variable. Afterwards, compute $\overline{h} = g \ominus h$ and add both h and $\overline{h}$ to the current partial solution.

Case (ii): Let h be the only candidate; compute $\overline{h} = g \ominus h$ and add it to the current partial solution.

Case (iii): A candidate haplotype is chosen with the same deterministic strategy described in Section 4.3.1. Its complement is calculated and added to the current partial solution.

Since we could not demonstrate whether all possible instantiation of the generalised phasing returned by the slave are equivalent, we also have to take into account the problem of evaluation. A possible solution is to take the values of random instantiations of the generalised phasing and compute a statistic, like the median or average, which will become our fitness value.

5.2 Algorithm for finding cliques in compatibility graph

We can learn one last thing from the running example introduced in the previous section. We report it for reference: find a maximally parsimonious phasing for $g_1 = 0120222$, $g_2 = 0222012$, $g_3 = 0100012$. When we were examining the resolution of g_1, we showed how a wrong choice of haplotype can lead to suboptimal solutions. One could also attempt to implement a heuristic that tries to resolve g_1 with a haplotype that is also compatible to a neighbor of g_1 in the compatibility graph. Indeed, this heuristic choice is performed by both algorithms presented in this thesis (see lower level resolution in Section 3.2.1 and the slave procedure in Section 4.3.1). Nevertheless, a suboptimal selection could still be performed; for instance, if we picked haplotype 0110011—compatible to both g_1 and g_2—the final solution would have cardinality 5 instead of 4. We certainly could extend this heuristic to include not only couples of compatible genotypes, but triplets; notice that such triplets correspond to triangles, or 3-cliques, in the compatibility graph. If we iterate this reasoning, it appears that it is convenient to find haplotypes compatible to all genotypes that form a clique in the compatibility graph.

To this aim, in this section we delineate an algorithm that finds all cliques in a compatibility graph and the set of haplotypes compatible to all genotypes in each clique.

To further stress the importance of finding cliques in the compatibility graph, we point out that a solution with value $k+1$ can be easily found for a k-clique of genotypes. Such solution would have a single haplotype with coverage k and k haplotypes with coverage 1 (for the definition of coverage see Section 2.3). This consideration might be important for tackling Haplotype Inference by Minimum Entropy. Moreover, finding haplotypes that resolve a whole clique could be an effective way to bootstrap Clark's Rule (Section 3.1).

5.2.1 Preliminary definitions

It is useful to introduce the concept of *schema*, taken straight from genetic algorithms, because it helps to generalise properties of haplotypes and genotypes. A schema s is a string in the alphabet $\Sigma = \{0, 1, *\}$ where $*$ is called wildcard. A m-length schema compactly represent binary strings of length m: if we instantiate in every possible way all wildcards in a schema, we obtain all strings it represents. The number of string a schema s represents is $2^{w(s)}$ where

$w(s)$ is the number of wildcards. For example, schema $s = 10**11$ compactly represents the set $\{100011, 100111, 101011, 101111\}$.

The resemblance of schemata to genotypes is clear: if we replace wildcards with 2's we can transform a schema in a genotype and *vice-versa*. Interestingly, a genotype is a compact representation of the set of haplotypes that can resolve it. Because of this property we can interchangeably use schemata and genotypes. In a similar way, a schema without wildcards is isomorphic to a haplotype.

As well as we did for genotypes, we can introduce compatibility between schemata, whose definitions is the same as Definition 3, just replacing 2's with wildcards. If we identify schemata as the sets they represent, we can say that two schemata are compatible if their set intersection is non-empty. We still write $s_1 \sim s_2$ to indicate compatibility. For example, these schemata are compatible $\{01**0*, *101*0, 0**10*\}$, while these are not $\{01*10, 0*1*1\}$.

Since schemata represent sets, it is useful to introduce an intersection operation. Schema intersection is defined only if operands are compatible and its definition is straightforward: the intersection of two compatible schemata s_1 and s_2, indicated as $s_1 \cap s_2$, is a schema whose i-th element is 0 (resp. 1) if the corresponding element in s_1 or s_2 is 0 (resp. 1), otherwise its a wildcard. [1] For example, $01*1** \cap *101* = 0101*$.

5.2.2 Algorithm for finding cliques in compatibility graph

The algorithm we are about to presents is able to find all cliques in a compatibility graph and, contemporarily, the haplotypes compatible to all genotypes in such cliques.

The rationale behind the algorithm is simple. Suppose that we have a compatibility graph $\mathcal{G}_c$; the idea is to construct another undirected graph $\mathcal{G}'_c$ whose vertices are schemata obtained in the following way: for every edge $(g_i, g_j) \in \mathcal{G}_c$, a vertex s' in $\mathcal{G}'_c$ is $g_i \cap g_j$.[2] Now every vertex in $\mathcal{G}'_c$ corresponds to a set of haplotypes that solve both g_i and g_j. In addition, $\mathcal{G}'_c$ has an edge (s'_i, s'_j) iff $s'_i \sim s'_j$: practically $\mathcal{G}'_c$ is a *higher-order* compatibility graph. If we iterate this procedure, we obtain compatibility graph of further higher orders whose vertices represent sets of haplotypes compatible to potentially many genotypes.

In the following we formalise the algorithm, demonstrate that converges and that finds all cliques in the compatibility graph.

Description of the algorithm. Let $G = \{g_1, \ldots, g_n\}$ be a genotype set. The initial step of our algorithm (iteration $l = 1$) consists in building an auxiliary graph data structure $\mathcal{G}_s^1$[3] which is a variant of the compatibility graph $\mathcal{G}_c$: its vertices are the schemata $\{s_1^1, \ldots, s_n^1\}$, each one corresponding to its homologous genotype g_i, it has an edge (s_i^1, s_j^1) iff $s_i^1 \sim s_j^1$ and each vertex is associated to a set $\sigma(s_i^1) = \{g_i\}$. The generic step at the $l+1$-th iteration is described in what follows. Let $\mathcal{G}_s^l$ be our graph data structure built at the previous iteration. If $\mathcal{G}_s^l$ contains only isolated vertices (every vertex has degree 0) the algorithm stops; otherwise produces a new graph $\mathcal{G}_s^{l+1}$ and continues. $\mathcal{G}_s^{l+1}$ has a vertex $s^{l+1} = s_i^l \cap s_j^l$ for every edge $(s_i^l, s_j^l) \in \mathcal{G}_s^l$ (duplicates are eliminated) whose

[1] Compatibility prerequisite excludes cases where an operand has a 0 and the other a 1 in the i-th position.

[2] Here genotypes are interpreted as schemata.

[3] In the following description, superscripts denote iteration counters.

associated set is $\sigma(s^{l+1}) = \sigma(s_i^l) \cup \sigma(s_j^l)$. $\mathcal{G}_s^{l+1}$ has also an edge (s_i^{l+1}, s_j^{l+1}) iff $s_i^{l+1} \sim s_j^{l+1}$.

The algorithm returns a list of L graphs $\mathcal{G}_s^1, \ldots, \mathcal{G}_s^L$ one for each iteration performed.

Cliques and haplotypes. By construction, zero-degree vertices in any $\mathcal{G}_s^l, 1 \leq l \leq L$ correspond to *maximal cliques* in the compatibility graph. Let us call s^l one of these vertices; $\sigma(s^l)$ contains the genotypes in the clique and $|\sigma(s^l)|$ is, of course, its order; in addition, the set described by s^l contains all haplotypes that are compatible to the genotypes in $\sigma(s^l)$. Finally, $\mathcal{G}_s^L$ is a graph whose vertices have null degree; its vertices correspond to *maximum cliques* on $\mathcal{G}_s^1$ (or $\mathcal{G}^c$ which is the same).

Termination and complexity. The algorithm does not diverge because the genotype set G is finite: during an iteration the algorithm calculates, at most, a set $\sigma(\cdot)$, and therefore a vertex, for every non-empty subset of G. The number of vertices computed is thus bounded from above by $2^{|G|} - 1$. Moreover, the algorithm does indeed converge. By construction, $\mathcal{G}_s^L$ contains, in fact, a vertex for every maximum clique in $\mathcal{G}_c$: since cliques are finite in number and their order is finite, the algorithm must converge. The complexity of the algorithm is, in the worst case, exponential (of course this algorithm can solve the $\mathcal{NP}$-complete Maximum Clique Problem which has worst-case exponential complexity). A worst-case instance can be generated this way: $G = \{g_1, g_2, \ldots, g_n\}$ where genotype g_i has only heterozygous sites except the i-th which is a 0; this way we obtain a complete graph and exponentially many cliques.

5.3 Conclusions and discussion

In this chapter we described two novel approaches to the Haplotype Inference problem that have the potential to improve the performance of the metaheuristics described in previous chapters. In Section 5.1 we provided an extension to the plain binary haplotype model that integrates CP techniques. This model, based on generalised haplotypes, addresses the restrictiveness of early commitment and the inadequacy of the usual definition of complementarity when dealing with unknowns. Section 5.2 presents a worst-case exponential algorithm able to find all cliques in the compatibility graph of an instance. Such cliques correspond to (ordinary) haplotypes which have high coverage and therefore can represent a good starting point for constructive techniques, which include also the simple Clark's Rule. Although this algorithm is exponential, we conjecture that it has much lower complexity in typical, real-world cases.

Chapter 6

The Founder Sequence Reconstruction Problem

In this chapter we introduce a new problem in computational genomics called Founder Sequence Reconstruction (FSRP). This problem is related to Haplotype Inference and represents, as we will see, a natural next step in association studies.

Similarly to what we did in previous chapters for the Haplotype Inference Problem, we formulate the FSRP as a combinatorial optimization problem which takes into account, by imposing suitable constraints and objective function, a particular genetic model.

In this chapter we will present two local search algorithms. The first one belongs to the family of Iterated Greedy algorithm while the second is an example of Large Neighbourhood Search metaheuristic. The two algorithms have been designed with different goals in mind. The former is simpler and able to return good solutions in limited time even for large-scale instances, but does not return high quality solution even in long runtimes. The latter, on the other hand, is more complex and sophisticated, but is often capable to find the optimum; moreover, if given enough time, is also able to return a provably optimal solution because it converges to a complete search. A promising idea could be to construct a good starting solution for the Large Neighbourhood Search with the former, faster Iterated Greedy. This happens to be a winning approach here because, as we will see, this last algorithm's behaviour is dependent on the initial solution quality; specifically there is statistical positive correlation between the initial and final solution values.

A common characteristic that both algorithms share with the other metaheuristics described in previous chapters, is that they embed and enhance existing procedures; specifically, the Iterated Greedy improves a stochastic greedy constructive, while the Large Neighbourhood Search exploits and enriches a complete solver. On a final note, these are two examples of the effectiveness of metaheuristics, whose capability of integrating different techniques is an important asset.

This chapter is principally based on material from the works by Benedettini et al. [18] (Sections 6.4 and 6.5 which present the Iterated Greedy algorithm and Section 6.6 with their experimental evaluation) and Roli et al. [184] (Sections 6.7

describes the Large Neighbourhood Search algorithm while Sections 6.8 and 6.9 are dedicated to its empirical evaluation).

6.1 Biological introduction and motivations

In recent years, the availability of biological data has rapidly increased as a consequence of technical advances in sequencing genetic material, such as DNA and haplotyped data. Given a sample of sequences from a population of individuals, researchers may try to study the evolutionary history of the population. If the population under study has evolved from a relatively small number of founder ancestors, the evolutionary history can be studied by trying to reconstruct the sample sequences as fragments from the set of founder sequences. The genetic material of the population individuals is the result of recombination and mutation of their founders. Many findings from biological studies support the validity of this model, as, for example, the fact that the '*Ferroplasma* type II genome seems to be a composite from three ancestral strains that have undergone homologous recombination to form a large population of mosaic genomes' (quoted from [225]). The main issue is that the number of founder sequences, as well as the founder sequences themselves, are generally unknown. A combinatorial optimisation problem can be defined such that its solutions are biologically plausible founder sequences, provided that assumptions on the evolutionary model are formulated. This problem is called the Founder Sequence Reconstruction Problem (FSRP) [227].

From this description, the FSRP can be regarded as a "natural continuation" of the Haplotype Inference in association studies. Indeed one can gather genotype data from a population, run a Haplotype Inference algorithm and then, after having obtained a biologically acceptable phasing, apply a reconstruction algorithm to compute a small set of ancestors.

Besides its origins in evolutionary biology, the problem is a hard combinatorial optimisation problem and it is of interest *per sé*, as it constitutes a challenge for current optimisation techniques.

In the literature, several techniques have been proposed to tackle this problem, such as dynamic programming, tree-search and metaheuristics. In this thesis, we present and analyse two algorithms for the FSRP: a fast randomised Iterated Greedy and a state-of-the-art hybrid algorithm that is based on the Large Neighbourhood Search framework. The former improves upon the simple constructive heuristic proposed in [185] in two ways: first, it incorporates look-ahead techniques into the constructive; secondly, it embeds the constructive into the Iterated Greedy framework [219], which is a generic metaheuristic based on the construction and partial destruction of solutions.

The latter performs a local search in which neighbourhoods have exponential size in the founder sequence length; the neighbourhoods are exhaustively explored by means of an efficient tree-search technique. Several variants of this hybrid search strategy are designed and compared on three benchmark sets. The best variant is compared to the state-of-the-art techniques and shown to achieve better performance.

This chapter is structured as follows. In Section 6.2, we formally introduce the problem. Next, we briefly survey previous and related work in Section 6.3. The following three sections are devoted to the Iterated Greedy algorithm: Sec-

Set of recombinants $\mathcal{C}$	Set of founders $\mathcal{F}$	Decomposition
01001000	01101110 (a)	a a|b|a a|c c c
00111000	10010011 (b)	a|c c c c c c c
10011100	10111000 (c)	b b b b|a a|c c
10111010		c c c c c c|a a
01101110		a a a a a a a a
10110011		c c c c|b b b b

Figure 6.1: On the left, a set of six recombinants in matrix form is shown. Assuming that the number of founders is fixed to 3, a valid solution as a matrix of three founders is shown in the centre. Denoting the first founder by a, the second founder by b, and the third one by c, on the right a decomposition of the recombinants matrix into fragments taken from the founders is shown. Breakpoints are marked by vertical lines. This is a decomposition with 8 breakpoints, which is the minimum value for this instance.

tion 6.4 details the constructive look-ahead procedure embedded in the Iterated Greedy; Section 6.5 introduces the Iterated Greedy framework itself in general terms and its application to the FSRP; Section 6.6 is dedicated to parameter configuration and experimental evaluation. The remaining sections address the Large Neighbourhood Search algorithm. In Section 6.7, we describe the Large Neighbourhood Search family of algorithms we designed and implemented; the experimental comparison among them is discussed in Section 6.8. In Section 6.9, the best of our Large Neighbourhood Search metaheuristic is compared with the state of the art.

6.2 The Founder Sequence Reconstruction Problem

The FSRP can be defined as follows. Given is a set of n *recombinants* $\mathcal{C} = \{C_1, \ldots, C_n\}$; each recombinant C_i is a string of length m over a given alphabet Σ, i.e., $C_i = c_{i1}c_{i2}\ldots c_{im}$ with $c_{ij} \in \Sigma$. In this work, we will consider a typical biological application where the recombinants are haplotyped sequences and, hence, $\Sigma = \{0, 1\}$. The symbols 0 and 1 encode the most common allele of the haplotype site (*wild-type*) and its most frequent variation in a population (*mutant*), respectively.

A *candidate solution* to the problem consists of a set of k_f *founders* $\mathcal{F} = \{F_1, \ldots, F_{k_f}\}$. Each founder F_i is a string of length m over the alphabet Σ: $F_i = f_{i1}f_{i2}\ldots f_{im}$ with $f_{ij} \in \Sigma\ \forall\, j$. A candidate solution $\mathcal{F}$ is a *valid solution* if the set of recombinants $\mathcal{C}$ can be *reconstructed* from $\mathcal{F}$. This is the case when each $C_i \in \mathcal{C}$ can be decomposed into a sequence of $p_i \leq m$ fragments (i.e., strings) $Fr_{i1}Fr_{i2}\ldots Fr_{ip_i}$, such that each fragment Fr_{ij} appears at the same position in at least one of the founders. Hereby, a decomposition with respect to a valid solution is called *reduced* if two consecutive fragments do not appear in the same founder. Moreover, for each valid solution $\mathcal{F}$ we can derive in $O(n \cdot m \cdot k_f)$ time (see [240]) a so-called *minimal decomposition*. This is

a decomposition where $\sum_{i=1}^{n} p_i - m$ is minimal. In the following we call this number the objective function value of $\mathcal{F}$ and denote it by $f(\mathcal{F})$. In biological terms, $f(\mathcal{F})$ is called the number of *breakpoints* of $\mathcal{C}$ with respect to $\mathcal{F}$.

The optimisation goal considered in this dissertation is to find a valid solution $\mathcal{F}^*$ that, given a fixed number of founders k_f, minimises the number of breakpoints. For an example, see Figure 6.1.

The FSRP was first proposed by Ukkonen [227] and the problem is proven to be $\mathcal{NP}$-hard [29, 178] for $k_f > 2$.

6.3 Overview of the literature

This section provides an overview of the scientific literature on the FSRP. First we discuss the methods devised to tackle the FSRP. Subsequently, other problem variants will be mentioned. Particular relevance will be given to RecBlock because of its strong connection to the algorithms we present in this work. Another closely related approach is discussed in [185] in which a Tabu Search method is proposed.

An algorithm based on dynamic programming was first proposed by Ukkonen [227]. However, this method is not efficient when the number of founders and the number or length of recombinants is high. Another dynamic programming algorithm was introduced by Rastas and Ukkonen [178]. Lyngsø and Song [149] have proposed a Branch-and-Bound algorithm. Although promising, this method has been evaluated only on a limited test set composed of rather small instances.

Wu and Gusfield have proposed a constructive tree-search algorithm [240]—henceforth referred to as RecBlock—which can be run as an exact or incomplete method. A more detailed explanation of RecBlock is provided in Section 6.3.1.

Many authors have provided contributions to upper and lower bound computations. Meyers and Griffiths have developed the R_h and R_s lower bounds [161], which are tighter than the one introduced by Hudson and Kaplan [116]. Nevertheless, they have worst case exponential and super-exponential complexity, respectively [8]. In the same paper, Meyers and Griffiths presented a general framework, called *composite method*. It exploits local bound information, i.e., a bound on a subset of contiguous columns in a recombinant matrix, in order to obtain a possibly better overall bound estimation. Bafna and Bansal suggest a lower bound computable in $O(n \cdot m^2)$ time [8]. Song et al. further elaborate on Meyers and Griffiths bound and propose improved lower and upper bounds [213]. In another work, Wu [239] developed an analytical upper bound which corresponds to an estimation of the minimum number of breakpoints independent of the particular recombinant set. This estimation is a function of only n and m.

The FSRP is widely studied and it is related to a number of other problems. El-Mabrouk and Labuda [71] focus on a much simpler and tractable reconstruction problem: given a founder set and a recombinant, they want to find a minimal decomposition for the recombinant. A similar problem, called *Haplotype Colouring*, has been introduced by Schwartz et al. [195]. Given a recombinant matrix, a partition of contiguous sites, called blocks, is identified according to some rule and each recombinant is subdivided accordingly into haplotype sub-

strings. The objective is to assign, for each block, a different colour to the distinct haplotype substrings in a block, such that the total number of colour switches between two contiguous haplotype substrings in the same recombinant is minimal. This problem of colouring each block can be encoded into an instance of the weighted bipartite matching problem. Haplotype Colouring can be solved to optimality by a $O(m \cdot n^2 \cdot \log n)$ algorithm.

The FSRP formulation we employ in this work is based on the assumption that, during the genetic evolution of the initial population of founders, mutation events are unlikely to occur. Rastas and Ukkonen generalise the FSRP by introducing a different objective function that takes into account point mutation [178], namely $f(\mathcal{F}) + c \cdot g(\mathcal{F})$ where $c > 0$ is a constant. This new objective function is the sum of two contributions: $f(\mathcal{F})$ denotes the sum of the number of breakpoints across all recombinant sequences (i.e., the objective function we introduced in Section 6.2), while $g(\mathcal{F})$ is the total number of point mutations. However, evolutionary biology researchers usually neglect mutations, as they are rare events. Therefore, in this work we focus on the version of the problem without mutation, which is equivalent to the generalised version if c is set to a very large value.

Zhang et al. [248] present a problem closely related to the FSRP called the *Minimum Mosaic Problem*. This problem aims at finding a minimum mosaic, i.e., a block partitioning of a recombinant set such that each block is *compatible* according to the Four Gamete Test [116] and the number of blocks is minimised. Differently from the FSRP, this problem does not rely on the existence of a founder set. This mosaic structure so obtained provides a good estimation of the minimum number of recombination events (and their location) required to generate the existing haplotypes in the population, i.e., a lower bound for the FSRP.

6.3.1 RecBlock

RecBlock is a tree-search based technique for the FSRP. It follows a constructive breadth-first search strategy starting from an empty solution. Each node ν_l at depth l in the search tree represents a feasible partial solution $\mathcal{F}_l$ up to the l-th site, i.e., a $k_f \times l$ founder matrix. As a consequence, complete solutions are at a tree depth equal to m. In other words, RecBlock fills a founder matrix column by column from left to right. Each node ν_l is labelled with the number of breakpoints in the minimal decomposition of $\mathcal{C}_l$, the problem instance up to site l, with respect to the associated founder matrix $\mathcal{F}_l$. We will denote this number by $BP(\nu_l)$. In the following, ν will interchangeably refer either to a node of the search tree or to the (partial) solution associated to ν.

In order to prune the search tree that would result from a naïve enumeration, RecBlock exploits symmetry breaking, bounding and dominance rules. Symmetry breaking relates to the fact that, in a solution to a FSRP instance, the founder sequence order is irrelevant. That is, any permutation of the order of the founders represents the same solution. Therefore, RecBlock constructs only solutions whose founders are sorted lexicographically. Bounding follows the usual procedure of classical Branch-and-Bound algorithms: a node is not expanded any further if its lower bound is greater than a known upper bound. The initial upper bound is obtained by running the incomplete version of RecBlock (explained below); the lower bound employed is computed by

the *composite bound method* [161]. The authors of RecBlock state that this bounding strategy is not very effective because the composite lower bound is not very tight. Much more effective is to prune *dominated* nodes. Let us consider two nodes at level l, ν_l^j and ν_l^h: if $BP(\nu_l^j) - BP(\nu_l^h) \geq n$, then node ν_l^j is dominated by ν_l^h and can be pruned. The reason is that any completion of the partial solution ν_l^j can be obtained starting from ν_l^h with the introduction of at most n breakpoints. This pruning criterion can be sharpened by computing the number n' of common founders between the decompositions of $\mathcal{C}_l$ corresponding to nodes ν_l^j and ν_l^h and pruning ν_l^j in case $BP(\nu_l^j) - BP(\nu_l^h) \geq n - n'$. Further details about this pruning mechanism can be found in [240].

RecBlock can also perform an incomplete search, in which nodes are pruned by applying a heuristic criterion. This heuristic is based on a score computed as the number of breakpoints in the partial solution associated to a node. The nodes that have a score that differs from the score of the current best node by more than a given threshold are discarded. In addition, the maximum number of search paths along the tree can also be limited. The resulting algorithm performs a breadth-first search limited to a part of the whole search tree and returns one solution—not proven to be optimal—only when this partial tree has been completely explored. The advantage of this version of RecBlock (henceforth called RecBlock-incomp) is that a feasible solution can be returned also to some large-size instances that are beyond reach for the complete version; on the other side, it has the disadvantage that it still might require extremely high execution times to return a solution and it is not able to improve the solution returned if more time is available.

RecBlock can be also perform an more lightweight incomplete search than the one executed by the RecBlock-incomp version. We call this version of RecBlock RecBlock-D0C1—the label is due to the command arguments passed to RecBlock executable, i.e., -D0 -C1—and essentially is a greedy constructive. RecBlock-D0C1 fills empty columns in a partial solution one after the other in a greedy fashion, that is, by choosing among all possible feasible binary strings of length k_f the one that minimises the current number of breakpoints. As we will see in Section 6.6.2, this version is very fast but returns poor quality solutions.

6.4 A Simple Constructive Heuristic With Look-Ahead

In the following we outline a randomized version of the simple constructive heuristic proposed in [185] extended by a look-ahead technique. This algorithm, which can be applied in a multi-start fashion as shown in Algorithm 4, is henceforth denoted by Greedy. In the following we explain in detail the working of function ComputeRandomizedSolution(). Note that throughout the presentation of our algorithms, the set of recombinants $\mathcal{C}$ is regarded a matrix with n rows and m columns. In the same way, a solution $\mathcal{F}$ is a matrix with k_f rows and m columns.

Initialization and filling of the first column. The solution construction process starts by filling the first column of $\mathcal{F}$, which is done as follows. First, the

Algorithm 4 GREEDY

1: INPUT: $nt \geq 1$, $lh \geq 1$, $rnd \in [0,1]$
2: $\mathcal{F} \leftarrow$ ConstructRandomizedSolution()
3: **while** termination conditions not met **do**
4: $\quad \mathcal{F}' \leftarrow$ ConstructRandomizedSolution()
5: $\quad$ **if** $f(\mathcal{F}') < f(\mathcal{F})$ **then** $\mathcal{F} \leftarrow \mathcal{F}'$ **endif**
6: **end while**
7: OUTPUT: $\mathcal{F}$

fraction p of 0-entries in the first column of $\mathcal{C}$ is derived. Then, two counters are introduced; counter n_0 for the 0-entries in the first column of $\mathcal{F}$, and counter n_1 for the 1-entries in the first column of $\mathcal{F}$. Both counters are initialized to 1 to ensure at least one 0-entry, respectively one 1-entry. Finally, a random number q from $[0,1]$ is drawn $k-2$ times. In case $q \leq p$ counter n_0 is incremented, n_1 otherwise. The first column is then composed of n_0 0-entries, followed by n_1 1-entries. After filling the first column, some data structures are initialized. For each row i of $\mathcal{C}$ a variable cp_i is kept that stores the position of the last breakpoint. These variables are initialized to 0, because no breakpoint exists yet. More specifically, $cp_i = 0$, for $i = 1, \ldots, n$. Moreover, a variable rep_i is kept that stores the index of the founder that represents row i of $\mathcal{C}$ after the last breakpoint cp_i. For all rows of $\mathcal{C}$ with a 0-entry in the first column this variable is initialized to 0, while for each row of $\mathcal{C}$ with a 1-entry the respective variable is initialized to $n_0 + 1$, that is, the first row of $\mathcal{F}$ with a 1-entry in the first column. More specifically, $rep_i = 0$ if $c_i = 0$, and $rep_i = n_0 + 1$ otherwise.

Filling of a column. After filling the first column of $\mathcal{F}$ and the initialization of the data structures, solution $\mathcal{F}$ is completed iteratively by filling one column after another. In the following we first outline the mechanism without look-ahead procedure. Let us assume that the first $j-1$ columns are already filled, and let us denote the corresponding partial solution by $\mathcal{F}_1^{j-1}$. Accordingly, the current column to be filled is column j. The positions of column j are filled one after the other, starting from row 1. To fill position f_{ij}, let n_0 be the number of rows of $\mathcal{C}$ that are represented by founder i and that have a 0-entry in position j. More specifically, n_0 is the number of rows r of $\mathcal{C}$ with $rep_r = i$ and $c_{rj} = 0$. Correspondingly, n_1 is the number of rows r of $\mathcal{C}$ with $rep_r = i$ and $c_{rj} = 1$. The actual setting of f_{ij} depends on parameter $rnd \in [0,1]$ that determines the amount of stochasticity that is introduced into solution construction. A random number q is drawn uniformly at random from $[0,1]$. If $q < rnd$, f_{ij} is set to 1 with probability $\frac{n_1}{n_1+n_0}$, and to 0 otherwise. If $q \geq rnd$, f_{ij} is set to 1 in case $n_1 > n_0$, and $f_{ij} = 0$ in case $n_0 > n_1$. Otherwise (that is, in case $n_0 = n_1$) a value for f_{ij} is chosen uniformly at random. This means that, even when $rnd = 0$, there is still some randomness in the solution construction, introduced by cases where $n_0 = n_1$.

If, after assigning a value to f_{ij}, row i can not be represented anymore by its current representative, one may try to change its representative by an equally good one. In case $f_{ij} = 0$, this concerns all rows r of $\mathcal{C}$ with $rep_r = i$ and $c_{rj} = 1$; similarly in case $f_{ij} = 1$. For all these rows r of $\mathcal{C}$ a new representing founder l (where $i < l \leq k$) that can equally represent r starting from breakpoint cp_r

Algorithm 5 General Iterated Greedy algorithmic framework

```
1: s ← GenerateInitialSolution()
2: while termination conditions not met do
3:    s_p ← Destruction(s)
4:    s' ← Construction(s_p)
5:    s ← AcceptanceCriterion(s, s')
6: end while
7: OUTPUT: best solution found
```

is searched, that is, a row l in $\mathcal{F}$ (where $i < l \leq k$) such that $c_{rs} = f_{ls}$, for all $s = cp_r, \ldots, j-1$. In case such a founder l can be found, rep_r is set to 1, and the search for an alternative representative for row r is stopped.

As a last step, after filling all the positions of column j, the variables cp_r and rep_r must be updated for all rows r of $\mathcal{C}$ for which $f_{rep_r j} \neq c_{rj}$. In such a case, the founder i with the minimum l such that $c_{rs} = f_{is}$, for all $s = l, \ldots, j$ must be determined. After identifying such a founder i, cp_r is set to 1, and rep_r is set to i.

For further reference, let us call this column-filling heuristic RGREEDY.

Look-ahead variant. The look-ahead variant of GREEDY depends on parameters nt (the number of trials) and lh (the look-ahead size). Let us assume that a partial solution $\mathcal{F}_1^{j-1}$ is given, which means that column j must be filled. For that purpose, nt matrices $\{J_1, J_2, \ldots, J_{nt}\}$ each one composed of k_f rows and $\min\{lh, m-j\}$ columns are generated by repeatedly applying RGREEDY. Note that each matrix J_i represents a possible extension of $\mathcal{F}_1^{j-1}$ by $\min\{lh, m-j\}$ columns. For each matrix J_i the optimal number of breakpoints bps_i obtained by appending J_i to the partial solution $\mathcal{F}_1^{j-1}$ is computed. Let $I = \arg\min\{bps_i\}$. The column to be appended to the partial solution $\mathcal{F}_1^{j-1}$ is then selected to be the first column of J_I.

6.5 A Probabilistic Iterated Greedy Algorithm

Several examples from the literature have shown that constructive heuristics may be improved by a simple metaheuristic framework known as an Iterated Greedy (IG) algorithm; see, for example, [188, 189, 219]). An IG algorithm starts with a complete solution. While some termination conditions are not met, it iteratively alternates between partial destruction of the incumbent solution (destruction phase) and re-construction of the resulting partial solution in order to obtain again a complete solution (construction phase). The general pseudo-code is provided in Algorithm 5.

The idea for our IG algorithm for the FSRP is based on the fact that solutions to a problem instance can be constructed from left to right, as explained in the previous section, but also from right to left: this is an intrinsic property (symmetry) of the FSRP. Based on this idea we developed the IG algorithm that is pseudo-coded in Algorithm 6 and henceforth denoted by BACKFORTH. In the following, the details of this algorithm are explained in depth.

Algorithm 6 BACKFORTH: An Iterated Greedy for the FSRP

```
1: INPUT: nt ≥ 1, lh ≥ 1, rnd ∈ [0,1], d ∈ [0,0.5], r ≥ 1
2: F ← ConstructRandomizedSolution()
3: right ← TRUE
4: while termination conditions not met do
5:    count = 0
6:    improved = FALSE
7:    while count < r and improved = FALSE do
8:       d_c = ⌊d · m⌋
9:       while d_c ≤ ⌊(1 − d) · m⌋ and improved = FALSE do
10:         F_p ← Destruction(F, d_c, right)
11:         F' ← ReconstructRandomizedSolution(F_p, d_c, right)
12:         if f(F') < f(F) then F ← F', improved ← TRUE endif
13:         d_c ← d_c + 1
14:      end while
15:      count ← count + 1
16:   end while
17:   right ← not right
18: end while
19: OUTPUT: F
```

Function ConstructRandomizedSolution() uses the constructive heuristic outlined in Section 6.4 for generating an initial solution from left to right, that is, it generates an initial column and then completes a feasible solution by applying RGREEDY. In the main loop, the algorithm tries to improve upon the current solution $\mathcal{F}$ either by removing columns from the right, or by removing columns from the left. This is done in function Destruction$(\mathcal{F}, d_c, right)$, where *right* is a Boolean variable that controls the switch between both variants. More specifically, when $right =$ TRUE columns are removed from the right hand side, and from the left hand side otherwise. The number of columns that are removed in function Destruction$(\mathcal{F}, d_c, right)$ is controlled by a parameter $d \in [0, 0.5]$. The actual number d_c of columns to be removed is first set to $\lfloor d \cdot m \rfloor$. If this does not prove to be successful, d_c is incremented until an upper limit of $\lfloor (1-d) \cdot m \rfloor$ is reached. This procedure is repeated at most $r \geq 1$ times, where r is another parameter of the algorithm. The usage of function Destruction$(\mathcal{F}, d_c, right)$ produces a partial solution $\mathcal{F}_p$. Commencing from this partial solution, function ReconstructRandomizedSolution$(\mathcal{F}_p, d_c, right)$ produces a complete solution $\mathcal{F}'$, employing the constructive heuristic RGREEDY outlined in Section 6.4. In case the newly produced solution is better than the current solution, variable *right* flips and the algorithm tries to improve the current solution from the other side.

6.6 Iterated Greedy experimental evaluation

Algorithms GREEDY and BACKFORTH were implemented in C++, compiled with `GCC` 3.4.6 and options `-O3 -fno-rtti -fno-exceptions` enabled. All experiments were performed on a cluster composed of dual core 2GHz Intel Xeon^{TM} processors with 6Gb of cache and 8Gb of RAM. As it is useful to distinguish between the randomized algorithm versions (that is, when $rnd > 0$) and

the quasi-deterministic algorithm versions (that is, when $rnd = 0$) we henceforth refer to the randomized versions as GREEDY-RND, respectively BACKFORTH-RND. Before we present a comparison of our algorithms to the state of the art, we first report on experiments that we performed in order to find a suitable parameter setting.

6.6.1 Parameter Tuning

For our experimentation we used the same benchmark set as introduced in [185]. This set is composed of randomly generated instances with $n \in \{30, 50\}$ recombinants and $m \in \{2n, 3n, 5n\}$ sites. More specifically, the benchmark set consists of five instances per combination of n and m. The generated instances are valid and not reducible, that is, no columns can be removed without affecting the optimal solution. Concerning our four algorithm types, that is, GREEDY, GREEDY-RND, BACKFORTH, and BACKFORTH-RND, we need to determine the best parameter values for each algorithm type. Moreover, we would like to infer how a certain parameter affects the behaviour of an algorithm type. From now on we refer to an algorithm type coupled with a specific parameter setting as an *algorithm instantiation*.

In the following we remind the reader about the relevant parameters of the different algorithms. First, all algorithms need a value for 1. the number of trials (nt), 2. the look-ahead size (lh), 3. and the amount of randomness used in the solution construction (rnd). In addition, BACKFORTH and BACKFORTH-RND algorithms requires a setting for parameters d (which controls the number of columns to be removed in the destruction phase) and r (the number of rounds for trying to improve the incumbent solution). Regarding nt and lh, we tested the following combinations: $(1, 1)$, $(5, 1)$, $(5, 2)$, $(5, 5)$, $(10, 1)$, $(10, 2)$, $(10, 5)$. The amount of randomness, rnd, may be selected from $\{0.0, 0.01, 0.1, 0.2\}$. For what concerns parameter d we allowed the following settings: $d \in \{0.1, 0.25, 0.4\}$. Finally, parameter r was set to 5 after initial experiments. These options result in 72 different algorithm instantiations. For the tuning experiments we selected 12 problem instances, one for each combination of n and m, as a *training set*. We applied each algorithm instantiation 10 times to each instance of the training set, for $k_f \in \{3, 5, 7, 10\}$. For $k_f = 3$ we excluded the combinations of with $nt > 5$ since the number of possible columns for a 3-founder solution is six. For each run we used a computation time limit of 50 seconds for the instances with 30 founders, and a computation time limit of 100 seconds for the ones with 50 founders.

For analyzing the results we employed a rank-based procedure as described in the following. In order to study whether the relative performance of the different algorithm instantiations depends on the number k_f of founders, we performed the same analysis for each number of founders. For each problem instance we computed the average over the 10 applications of each algorithm instantiation. Then, for each problem instance, we ordered the 72 algorithm instantiations according to the average they achieved, that is, the algorithm instantiation with the best average obtains rank 1, etc. Ties were given the same rank. Then, for each algorithm instantiation we computed the average rank by averaging over all instances of the training set. Afterwards, the 72 algorithm instantiations were ordered according to their average rank. This approach is particularly useful when dealing with problem instances where the objective function val-

Table 6.1: Rank-based analysis of the tuning experiments. The results are presented separately for each different number k_f of founders.

(a) Ranking for $k_f = 3$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	1.167	BackForth-Rnd	5	1	0.1	0.2
30	32.667	BackForth	5	5	0.1	
43	39.833	Greedy-Rnd	5	5		0.1
44	40.167	Greedy	5	5		

(b) Ranking for $k_f = 5$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	1.500	BackForth-Rnd	10	1	0.1	0.2
40	37.000	BackForth	10	5	0.1	
72	60.000	Greedy-Rnd	10	5		0.01
74	61.333	Greedy	10	5		

(c) Ranking for $k_f = 7$.

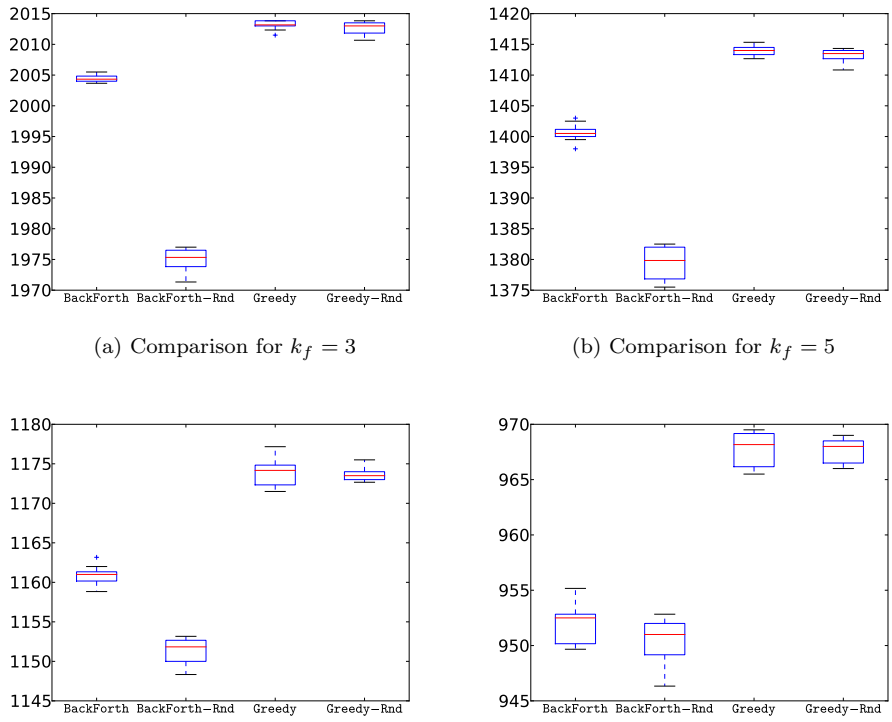
Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	1.667	BackForth-Rnd	10	1	0.1	0.2
17	18.167	BackForth	10	5	0.4	
68	53.000	Greedy-Rnd	10	5		0.1
70	54.167	Greedy	10	5		

(d) Ranking for $k_f = 10$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	3.167	BackForth-Rnd	10	1	0.1	0.2
3	6.833	BackForth	10	2	0.1	
64	50.833	Greedy	10	2		
67	51.667	Greedy-Rnd	10	5		0.01

ues are in different scales, like in our case. In Table 6.1 for each algorithm type (that is, Greedy, Greedy-Rnd, BackForth, and BackForth-Rnd) and each founder we report 1. the position of the first occurrence in the final ranking of the algorithm instantiations, 2. its average ranking on all problem instances, and 3. its description consisting of algorithm type and relevant parameter settings. These figures clearly demonstrate that the best performing algorithm instance, on average, is BackForth-Rnd with $nt = 10$ (5 in case $k = 3$), $lh = 1$, $d = 0.1$, and $rnd = 0.2$. In other words, the iterated greedy algorithm has clear advantages over the multi-start heuristic. Moreover, when algorithm BackForth is concerned, randomness is much more useful than in the case of algorithm Greedy.

In order to show also the qualitative difference between the four algorithms shown in Table 6.1, we compare the average solution qualities they achieved in Figure 6.2. This is done as follows. For each of the 10 applications we summed up the result achieved on all problem instances from the test set. This provides us with 10 values for each of the four algorithm instantiations. These 10 values are shown in the form of boxplots for each algorithm instantiation. The graphics



(a) Comparison for $k_f = 3$

(b) Comparison for $k_f = 5$

(c) Comparison for $k_f = 7$

(d) Comparison for $k_f = 10$

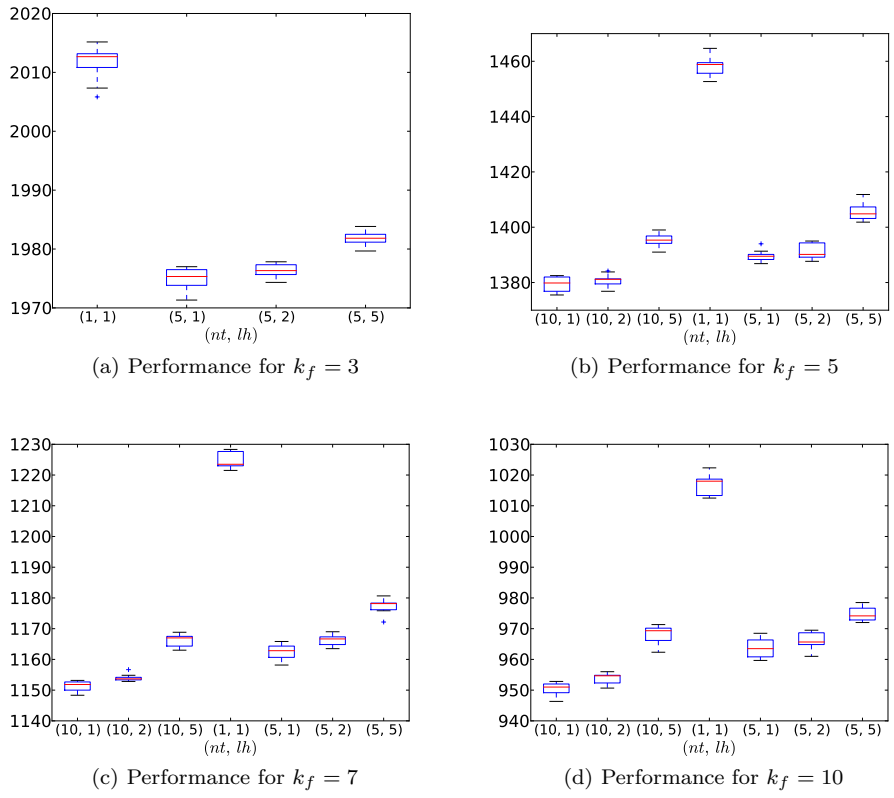
Figure 6.2: A comparison regarding the solution quality of the best algorithm instantiations of the four algorithm types.

clearly support the conclusions that we have drawn from the results of Table 6.1.

Finally, note that the best algorithm instantiation in the comparison does not use look-ahead (that is, $lh = 1$). This seems counter-intuitive at first, especially because the best performing algorithm instantiations of the other three algorithm types all use a look-ahead value greater than one. The reason seems that, given a limited amount of time, instead of looking ahead it is better to explore as many solutions as possible. The graphics of Figure 6.3 support this claim. Boxplots show the performance of the best instance of BackForth-Rnd on the whole training set over the 10 runs when varying the values of the (nt, lh) parameters. The y-axis reports the average solution value. As shown in the graphics, incrementing the look-ahead is detrimental to the algorithm performance, while incrementing the number of trials is beneficial. When $k_f = 10$ this phenomenon can also be observed for other algorithm instantiations as shown in Table 6.1d.

6.6.2 Comparison with the State of the Art

We tested the best algorithm instantiation as determined by the parameter tuning phase—henceforth denoted by Best—against all techniques used in [185]. For consistency reasons we maintain the same algorithm notifiers in the result



(a) Performance for $k_f = 3$

(b) Performance for $k_f = 5$

(c) Performance for $k_f = 7$

(d) Performance for $k_f = 10$

Figure 6.3: A comparison of algorithm instantiations of BackForth-Rnd with varying values of nt and lh.

tables as used in [185]. This means that *heuristic* actually refers to Greedy with $nt = 1$, $lh = 1$, and $rnd = 0$. Moreover, *TS* refers to the tabu search presented in [185]. The remaining algorithms are three variants of RecBlock: 1. complete version (RecBlock-comp), 2. incomplete variant (RecBlock-incomp), and 3. the lightest heuristic version (RecBlock-D0C1).

Remember that the benchmark set consists of 60 problem instances as outlined at the beginning of Section 6.6.1. Each instance was considered in combination with different numbers of founders, more specifically, we considered $k_f \in \{3, \ldots, 10\}$. Then, as a first experiment we applied Best for one hour to each combination of an instance and a founder number k_f exactly once. Results are summarized in Tables 6.2 and 6.3 in which the average solution qualities and the corresponding standard deviations are reported. Statistics are taken over the five instances per combination of the number of recombinants (n) and sites (m). Note that the results in [185] were also obtained with a computation time limit of one hour for each run. Even though the results from [185] were obtained on different processors, they are comparable because the processors have a similar speed.[1]

[1]Consider that all algorithms implemented in this paper are single-threaded and do not take advantage of parallel architectures; for what RAM is concerned, the algorithms in this

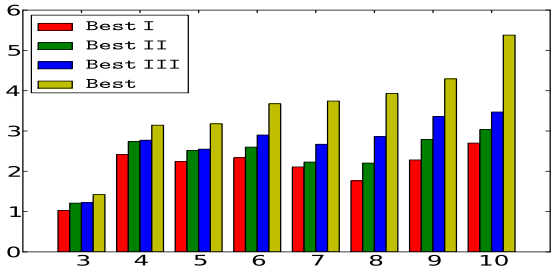
The results show that our algorithm achieves, in each case, a better performance than RecBlock-D0C1, *heuristic*, and *TS*. This is remarkable, because *TS* is build upon a sophisticated neighborhood structure, whereas our randomized iterated greedy algorithm is very un-sophisticated in comparison. For what concerns the comparison to *rec-exact* and RecBlock-incomp, our algorithm is generally inferior. However, *rec-exact* and RecBlock-incomp fail rather soon (that is, with growing problem size) to produce any solution within the allotted time of one CPU hour.

In order to study the development of the solution quality obtained by our algorithm over time, we run the algorithm also with other computation time limits. In particular, we were interested in the behaviour of our algorithm when computation time limits are much more restrictive. Table 6.4 shows three additional computation time limits that we used. The resulting algorithm instantiations are denoted by Best I, Best II, and Best III. In Figure 6.4 we show the results in the following form. Results are averaged over all instances with the same number of recombinants. For each different founder number we show the results of the Best with the four different computation time limits in terms of the percent deviation with respect to the results achieved by *TS*. First, it is interesting to note that in all cases the percent deviation is positive, which means that with all tested computation time limits our algorithm is better than *TS*. This is especially remarkable for the shortest computation time limits of 50 seconds per run for instances with 30 recombinants and 100 seconds per run for instances with 50 recombinants. Finally, as expected, the graphics show clearly that our algorithm improves with growing computation time limits.

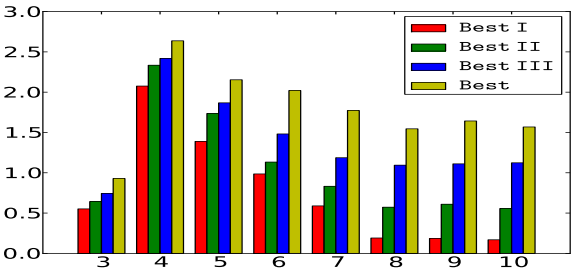
On the negative side, the obtained solution quality does not improve impressively over time. Therefore we conclude that, although Best is a valuable and scalable heuristic, it does not take the best possible advantage of larger time limits. That would suggest that our algorithm is especially suited either as a fast heuristic upper bound on medium- and large-sized problem instances or as an improvement step in the context of some population-based metaheuristic, such as a memetic algorithm or ant colony optimization.

These hypotheses are indeed true and the statistical analysis performed in Section 6.9.2 confirm that. This is the main reason why we combined our Iterated Greedy with a more sophisticated learning-based method; this way, as we will see in further sections, larger running times can be exploited much more beneficially.

paper use less than 5MB.



(a) Instances with $n = 30$



(b) Instances with $n = 50$

Figure 6.4: The y-axis shows the percent deviation of algorithms BEST I, BEST II, BEST III, and BEST over *TS*. Results are averaged over the instances with the same number of recombinants. The x-axis ranges over different numbers of founders.

Table 6.2: Results for instances with 30 recombinants. Results are averaged over 5 random instances. The symbol '—' indicates that no solution was returned. Standard deviations are reported in brackets; lowest entries are highlighted in boldface.

30 recombinants sites , founders	RecBlock-comp	RecBlock-incomp	RecBlock-D0C1	**heuristic**	**TS**	Best
60 , 3	**573.8 (12.38)**	579.4 (11.5)	604 (16.11)	594.2 (13.08)	583 (11.79)	574.6 (11.52)
60 , 4	**445.4 (5.59)**	450.2 (6.53)	494.2 (18.27)	479.6 (9.18)	459.6 (7.5)	448.2 (5.34)
60 , 5	—	385.2 (7.85)	425.4 (10.06)	412.2 (8.87)	395.8 (9.36)	**384.6 (7.42)**
60 , 6	—	340.6 (5.18)	383.6 (5.13)	367.6 (6.88)	352 (6.6)	**339.6 (6.34)**
60 , 7	—	**303.6 (5.64)**	353.8 (10.06)	335.2 (7.22)	318.2 (6.76)	306.6 (4.76)
60 , 8	—	**274.6 (3.71)**	331 (8.75)	311.6 (5.77)	291.2 (4.38)	281.8 (4.53)
60 , 9	—	—	307.4 (10.29)	288.6 (6.47)	270.4 (4.51)	**258.8 (6.49)**
60 , 10	—	—	294 (9)	268.4 (4.56)	251.8 (4.32)	**237.8 (5.71)**
90 , 3	**877.2 (2.95)**	885.2 (3.96)	917.8 (12.83)	910.8 (8.01)	892 (4.58)	879.6 (1.50)
90 , 4	**684.2 (3.27)**	689.4 (4.34)	749.4 (5.81)	741.6 (7.16)	711.8 (4.02)	690.0 (3.63)
90 , 5	—	**596.2 (4.49)**	653 (14.23)	645.6 (3.21)	618.6 (3.78)	600.2 (4.79)
90 , 6	—	**525 (2.45)**	584.2 (7.85)	580.2 (4.32)	552.8 (4.76)	532.8 (3.19)
90 , 7	—	**469.4 (3.91)**	542 (22.29)	529.8 (6.76)	500.4 (4.16)	482.4 (3.44)
90 , 8	—	**424.4 (2.7)**	498.8 (17.47)	491 (4)	461.2 (2.17)	444.4 (1.36)
90 , 9	—	—	469.8 (6.1)	456.2 (4.92)	427.8 (3.9)	**409.2 (2.40)**
90 , 10	—	—	438.2 (7.05)	427 (4.85)	398.8 (3.35)	**377.6 (3.38)**
150 , 3	**1468.8 (21.7)**	1482.6 (17.87)	1533.4 (16.46)	1529 (16.12)	1500.6 (18.65)	1480.0 (18.74)
150 , 4	**1140.4 (9.42)**	1154.4 (5.18)	1249 (18.72)	1253.2 (12.77)	1200.8 (10.76)	1157.6 (9.09)
150 , 5	—	**991.6 (8.2)**	1083.8 (20.68)	1090.8 (9.88)	1041.6 (10.78)	1005.8 (7.03)
150 , 6	—	**876.2 (6.26)**	971.2 (3.49)	980 (4.8)	932 (9.14)	899.4 (5.92)
150 , 7	—	—	888.8 (12.03)	897 (4.47)	848.2 (6.42)	**817.8 (3.43)**
150 , 8	—	—	819.2 (5.36)	831.8 (4.6)	783.2 (4.71)	**748.2 (5.42)**
150 , 9	—	—	770.2 (12.64)	773 (3.39)	727.6 (3.71)	**700.6 (4.76)**
150 , 10	—	—	715.2 (9.52)	724.8 (2.68)	676.6 (3.78)	**646.6 (4.59)**

Table 6.3: Results for instances with 50 recombinants. Results are averaged over 5 random instances. The symbol '—' indicates that no solution was returned. Standard deviations are reported in brackets.

50 recombinants sites , founders	RecBlock-comp	RecBlock-incomp	RecBlock-D0C1	**heuristic**	**TS**	Best
100 , 3	**1765.4 (16.96)**	1784.4 (14.64)	1837.8 (31.03)	1821.2 (18.02)	1789 (15.18)	1775.0 (14.57)
100 , 4	**1377.6 (10.88)**	1392.2 (9.39)	1481.8 (24.63)	1483.8 (8.23)	1425.2 (13.95)	1388.8 (11.23)
100 , 5	—	**1225.2 (14.72)**	1305 (17.36)	1301.2 (15.06)	1260.6 (14.43)	1232.2 (10.61)
100 , 6	—	**1095.8 (13.92)**	1177.6 (12.16)	1188.4 (15.08)	1140.2 (11.21)	1115.6 (12.67)
100 , 7	—	**997.8 (10.99)**	1087.8 (15.9)	1101.4 (9.89)	1049.4 (9.13)	1027.6 (10.33)
100 , 8	—	**920.4 (9.71)**	1026.8 (6.3)	1034.8 (9.78)	976 (9.62)	960.4 (9.54)
100 , 9	—	—	963.8 (14.82)	976.2 (13.59)	915 (11.73)	**897.0 (6.03)**
100 , 10	—	—	918.8 (6.76)	928.4 (10.64)	868 (8.34)	**851.2 (3.49)**
150 , 3	**2631.2 (22.88)**	2660.6 (22.74)	2740.8 (29.3)	2722.6 (23.99)	2677.4 (23.56)	2647.6 (22.92)
150 , 4	**2056.8 (5.72)**	2078.8 (6.91)	2194.2 (26.48)	2240.6 (6.88)	2148.2 (8.41)	2091.6 (10.48)
150 , 5	—	**1823.2 (8.32)**	1936.8 (12.74)	1965 (9.46)	1894.8 (8.35)	1857.2 (9.02)
150 , 6	—	**1635.8 (12.85)**	1759.6 (9.66)	1794.8 (6.8)	1717.8 (7.16)	1683.2 (12.67)
150 , 7	—	**1493.2 (11.19)**	1644 (12.53)	1668 (9.22)	1578.8 (10.18)	1554.4 (8.01)
150 , 8	—	—	1528.8 (13.24)	1562.8 (10.01)	1475.2 (10.96)	**1450.4 (5.12)**
150 , 9	—	—	1443.8 (6.69)	1479.2 (14.74)	1386 (8.86)	**1365.6 (7.91)**
150 , 10	—	—	1376.8 (15.59)	1403.2 (11.56)	1314.8 (5.81)	**1299.0 (6.03)**
250 , 3	**4421 (22.06)**	4466.2 (20.46)	4597.8 (33.69)	4601.6 (15.53)	4514.8 (11.95)	4475.8 (18.48)
250 , 4	**3448.67 (4.73)**	3490.8 (10.76)	3728.8 (8.53)	3813.6 (7.54)	3634.2 (13.88)	3542.6 (15.29)
250 , 5	—	**3071.4 (15.98)**	3258.4 (33.25)	3344 (21.12)	3218.8 (11.69)	3151.6 (22.97)
250 , 6	—	**2754.4 (14.17)**	2967.8 (24.77)	3046.8 (11.37)	2915.8 (17.31)	2864.2 (21.39)
250 , 7	—	**2510.6 (9.4)**	2735.6 (20.89)	2832 (13.82)	2686.6 (11.8)	2643.6 (11.46)
250 , 8	—	—	2570.6 (22.06)	2648.8 (17.77)	2504.8 (12.93)	**2472.6 (10.09)**
250 , 9	—	—	2422 (30.24)	2505.8 (14.79)	2358 (9.67)	**2324.8 (10.36)**
250 , 10	—	—	2304.4 (28.06)	2378.8 (7.22)	2237.2 (7.6)	**2203.8 (5.49)**

Table 6.4: Alternative computation time limits (in seconds).

Number of recombinants	Notifier		
	BEST I	BEST II	BEST III
30	50	100	180
50	100	200	600

Algorithm 7 High level algorithm of LNS for the FSRP

1: $\mathcal{F} \leftarrow$ BuildInitialSolution()
2: **while** termination condition not met **do**
3: $\quad\mathcal{I}_{\text{free}} \leftarrow$ ChooseFounders()
4: $\quad\mathcal{F}' \leftarrow$ RecBlock($\mathcal{F}|_{\mathcal{I}_{\text{free}}}$) {Solve the instance restricted to the set of free founders}
5: $\quad$**if** $f(\mathcal{F}') < f(\mathcal{F})$ **then**
6: $\quad\quad\mathcal{F} \leftarrow \mathcal{F}'$
7: $\quad$**end if**
8: **end while**
9: **output:** best solution found $\mathcal{F}$

6.7 Large neighbourhood search algorithms for the FSRP

The method that we propose is a hybrid metaheuristic [30] based on the Large Neighbourhood Search framework. Large Neighbourhood Search (LNS) is a search strategy consisting in a local search that uses a complete method for exploring a—typically very large—neighbourhood. LNS tries to combine the advantage of a large neighbourhood, that usually enhances the explorative capabilities of local search, with an exhaustive tree-search exploration which is faster than enumeration. This kind of search strategy has been successfully applied in several contexts [2, 7, 52, 174, 206]. We designed a family of algorithms based on the LNS framework which employ RecBlock [240] as sub-solver. In fact, RecBlock is rather efficient for a small number of founders and our aim was to exploit this fact for a method capable of tackling large-size instances. We first illustrate the high level search strategy we designed (Algorithm 7) and we subsequently detail the variants we implemented (Algorithm 8).

An initial solution is built by means of any constructive procedure (Line 1 of Algorithm 7), then a local search with large neighbourhoods is performed. Let $\mathcal{I}$ be the set of indices of founders in $\mathcal{F}$. The neighbourhood is composed of all feasible assignments to a selected set of founders corresponding to indices $\mathcal{I}_{\text{free}} \subseteq \mathcal{I}$ (Line 3), while the other founders corresponding to indices $\mathcal{I} \setminus \mathcal{I}_{\text{free}}$ are kept fixed; this neighbourhood is exhaustively explored by RecBlock (Line 4). For this purpose, we modified RecBlock so as to have the algorithm searching only among the configurations of $k_f = |\mathcal{I}|$ founders in which $|\mathcal{I} \setminus \mathcal{I}_{\text{free}}|$ founders are fixed. If the chosen neighbouring solution has a lower number of breakpoints than the current one, it becomes the new current solution, in the spirit of a first improvement local search (Lines 5–7) of Algorithm 7. This strategy shares also analogies with IG [110] and Forget and Extend [43], in that it partially destroys and reconstructs the solution.

Depending on the instantiation of the function ChooseFounders(), different versions of Algorithm 7 can be defined. We implemented several variants of the algorithm, which share the idea of increasing the number of unassigned founders whenever either all the combinations have been explored or no solution improvements are found for a given number of examined neighbours. We remark that the idea of enlarging the neighbourhood whenever diversification is needed is similar to that characterising Variable Neighbourhood Descent [106]. In our LNS

Algorithm 8 LNS-FSRP algorithm

```
1: Set parameters k_min, k_max, maxCombinations
2: F ← BuildInitialSolution()
3: k ← k_min
4: i ← 1
5: while k ≤ k_max ∧ maximum time not expired do
6:   if not all k-combinations of k_f founders explored ∧ i ≤ maxCombinations then
7:     I_free ← NextCombination(k, k_f)
8:     F' ← RecBlock(F|_{I_free})
9:     i ← i + 1
10:    if f(F') < f(F) then
11:      F ← F'
12:      k ← k_min
13:      i ← 1
14:    end if
15:  else
16:    k ← k + 1
17:    i ← 1
18:  end if
19: end while
20: output: best solution found F
```

algorithm, the neighbourhood size is controlled by the number k of founders to which the search is restricted. The high-level parametrised algorithm describing our family of algorithms is detailed in Algorithm 8.

The algorithm has three main parameters: k_{min} and k_{max}, which denote the minimum (resp. maximum) number of founders to be chosen, and $maxCombinations$, which denotes the maximum number of k-combinations of founders that are considered before incrementing k. The indices of a new k-combination of founders are returned by the function $\mathsf{NextCombination}(k, k_f)$. The neighbourhood is examined in random order, i.e., the combinations of indices are randomly picked from the set without replacement.

The search is stopped when either $k > k_{max}$ or the maximum computation time allotted is expired. The time check is also performed inside the modified version of RecBlock, so as to guarantee that the solution returned is found within the time limit.

Depending on the values of the parameters, different variants of LNS-FSRP can be obtained. The ones that we tested are the following:

- `LNS-1`: $k_{min} = 1$, $k_{max} = k_f$, $maxCombinations = \infty$
- `LNS-3`: $k_{min} = 3$, $k_{max} = k_f$, $maxCombinations = \infty$
- `LNS-maxc`: $k_{min} = 1$, $k_{max} = k_f$, $maxCombinations = 10$

`LNS-1` and `LNS-3` differ in the minimum number of founders considered for defining the neighbourhood. The advantage of `LNS-3` over `LNS-1` could be to start the search with a neighbourhood that makes it possible to achieve a good

balance between diversification and efficiency, because RecBlock has proven to be quite fast in solving instances with 3 founders. Note that the neighbourhoods can be explored exhaustively, because $maxCombinations = \infty$. Therefore, if no time limit is imposed, `LNS-1` and `LNS-3` both converge to a RecBlock search on the whole set of k_f founders; therefore, in principle, they are complete. As it will be shown in Section 6.9, this search strategy has the advantage of finding very good—or even optimal solutions—much faster than when running RecBlock directly in its full version; moreover, it finds good solutions to large-size instances, which are not solved be RecBlock. `LNS-maxc` differs from the previous algorithms in that it only performs a limited number of neighbourhood explorations; the rationale behind this strategy is that it could be beneficial, especially for large-size instances, to sample the k-combinations rather than enumerating all of them. $maxCombinations$ is a parameter of the algorithm and it should be set after a parameter tuning process. This parameter was set to 10, because this value is the minimum number that enables the algorithm to exhaustively explore the neighbourhood of size one in the cases in which $k_f = 10$, thus achieving a good trade-off between intensification and diversification across all the instances.

All three algorithms have the property of being *anytime algorithms*, because they progressively return improved solutions.

The initial solution can be provided by a dedicated constructive heuristic or it can be a random solution. In our algorithms we feed the LNS algorithm with a solution generated by the best out of the four algorithms, henceforth called B&F IG (Bestwould be a misleading label in the incoming algorithm tuning stage), coming from parameter configuration described in Section 6.6.1.

A possible improvement over the algorithms described above consists in exploiting the pruning capability of RecBlock when an upper bound (UB) is provided. Indeed, each time a new best solution is found, this value can be provided to RecBlock for pruning the search tree. The effect of the use of this UB update will be discussed in Section 6.8.

6.8 Large Neighbourhood Search experimental analysis

In this section, we first introduce the test instances. Subsequently, we experimentally compare the LNS variants. The best of these algorithms is compared to the current state-of-the-art method, RecBlock, and also to the B&F IG itself.

6.8.1 Experimental setting

The algorithms have been implemented in C++, compiled with `GCC` 3.4.6, options `-O3 -fno-rtti -fno-exceptions` enabled, and run on a cluster equipped with dual core Intel Xeon^{TM} 3GHz processors, 6MB of cache and 8GB of RAM. Since a short computation time is not a requirement for the FSRP, the maximum CPU time allotted for each algorithm was 3 days (72 hours). Whenever B&F IG was chosen for providing the initial solution, it was run for 50 seconds in the case of 30 recombinants and 100 seconds for instances with 50 recombi-

nants (confront with BEST I runtime settings from Table 6.4). The whole set of experiments required about 4 years of CPU time.

Algorithms are compared on the basis of solution quality and, in case of ties, the computation time at which the best solution of a run was found.

The benchmark instance set used in this study comprises three sets: `random`, `ms` and `evo`. The `random` set has been used also in previous works [185].A random instance is generated by assigning value 0 or 1 to each recombinant's site with probability equal to 0.5. All instances of the set `random` are valid and not reducible. Random instances have the disadvantage of not being particularly significant as benchmark for real-world problems. To compare the algorithms on instances of varying size with a structure similar to that of real-world instances, we generated two further sets based on evolutionary models. In fact, apart from the `random` instance set, in previous works only small-size real-world instances have been used. The first set is produced with the Hudson generator [115] that simulates genetic samples under neutral models. The second one is generated by simulating a simple neutral evolutionary process by iteratively applying a one-point crossover to an initial population of founders. Besides the number of sites (m), founders (k_0) and recombinants (n), the parameters of the `evo` generated instances are the crossover probability (cxp) and the growth factor (α). As long as the current population's size $popsize$ is less than n, a new population of size $\lfloor \alpha \cdot popsize \rfloor$ is generated by iteratively picking two individuals at random, producing two individuals by applying the crossover—with probability cxp—and inserting in the population randomly one of the two new individuals. The `evo` benchmark set was generated with $k_0 = 10, \alpha = 2, cxp = 0.8$. All instances in the `ms` and `evo` sets are valid and not reducible. Each set contains instances with n recombinants of length m, where $n \in \{30, 50\}$ and $m \in \{2n, 3n, 5n\}$. In previous works, values of k_f up to 5 were considered. With the aim of analysing the scaling behaviour of the algorithms, we considered values of founders up to 10. The instances used are available as online additional material [183].

It is important to observe that our techniques are stochastic, therefore they might return a different solution at each run. However, in our experiments, we evaluate each algorithm only once on a benchmark instance. We do so to avoid too high computation times that would be incurred by more than a single trial per instance. Moreover, it is known that for a fixed number N of runs the minimal variance estimation is achieved by running an algorithm once on N instances, instead of running it more than once on a smaller subset of instances [27]. In the comparisons which follow, whenever we need to assess the statistical significance of the results, we apply the non-parametric paired Wilcoxon test for the equality of two medians [57].

6.8.2 Comparison among LNS-FSRP variants

We first compare algorithms `LNS-1`, `LNS-3` and `LNS-maxc`. In order to save computation time, while keeping at the same time a wide spectrum of test cases, the problem instances are solved with $k_f \in \{5, 7, 9\}$; therefore, the algorithms were run on 54 cases (a case is an instance with a given k_f). Since solution values and execution times can be on different scale across the test cases, we compared the three algorithms on the basis of their relative difference per instance, computed with respect to the minimum among the three values. Let us denote by $sol(\mathcal{A})$ the solution value returned by algorithm $\mathcal{A}$ and let sol_{min} be the minimum

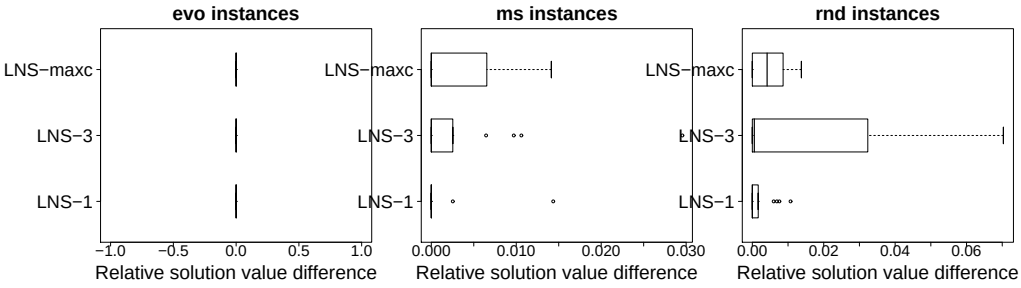


Figure 6.5: Boxplot of solution value differences between `LNS-1`, `LNS-3` and `LNS-maxc`.

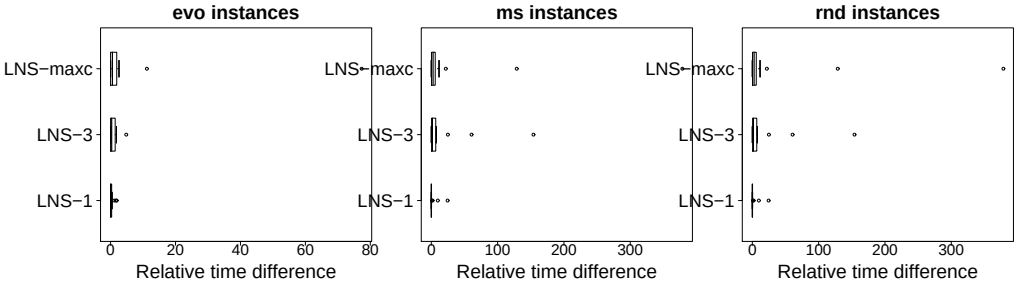


Figure 6.6: Boxplot of execution time differences between `LNS-1`, `LNS-3` and `LNS-maxc`.

value found by the algorithms involved in the comparison. The relative solution value difference of algorithm $\mathcal{A}$ is computed as $(sol(\mathcal{A}) - sol_{min})/sol_{min}$. Relative time differences are computed in an analogous way to the relative solution value differences. Boxplots of the relative solution value and the relative time differences are drawn in Figure 6.5 and Figure 6.6, respectively. Each boxplot graphically summarises statistics on the relative difference values attained by an algorithm across all the test cases. The bold line denotes the median value, while the leftmost and rightmost sides of the rectangle denote the 1st and 3rd quartile of the distribution, respectively.[2]

The results of this comparison show that the algorithms perform similarly, but that `LNS-1` is superior to the other two algorithms as it is statistically confirmed by the Wilcoxon test: `LNS-1` is significantly better than the other two competitors on `random` instances with respect to solution value and on `evo` and `ms` instance sets with respect to run time. No statistical difference is found in the other cases.

Before discussing the experimental results of the comparison against the state of the art, we present some further analysis on the behaviour of `LNS-1` by studying the impact of the initial solution and of a dynamic upper bound update. Furthermore, we analyse the impact of neighbourhood size on the

[2]For more details on boxplots, we refer the reader to [84].

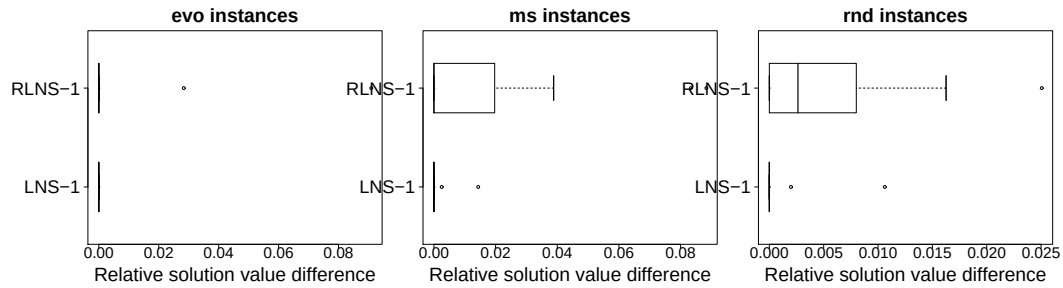


Figure 6.7: Boxplot of relative solution value difference between `LNS-1`, which is initialised by B&F IG, and `RLNS-1`, which is initialised by a random solution.

search. Finally, we present an improvement of the algorithm based on a speed-up mechanism.

6.8.3 Impact of initial solution

The algorithm we implemented starts the search from a greedy constructed solution provided by B&F IG. We also run experiments to assess the actual impact of a heuristic solution with respect to a random initial one. The corresponding algorithm is named `RLNS-1`. A summary of the statistics concerning the relative improvement of `LNS-1` over `RLNS-1` is shown in Figure 6.7. Except for the case of the `evo` instances, in which no statistically significant difference is found, starting from an initial solution provided by B&F IG enables the algorithm to find a better solution than starting from randomly generated founder sequences.

6.8.4 Impact of upper bound update

The complete algorithm RecBlock used in Line 8 of Algorithm 8 accepts as input also an upper bound value on the solution. This piece of information is used to prune the search tree during the solution process. Since we use iteratively RecBlock during the search, it might be beneficial to update the upper bound every time a new best solution is found. We implemented this variant of `LNS-1` and found no evidence for an advantage of using UB information over not using it, neither in solution quality nor in execution time.

6.8.5 Impact of neighbourhood size

During the runs of the algorithm, we recorded the neighbourhood size at which a new best solution was found, so as to estimate the impact of neighbourhood size on the search process. Data were collected on all instances and for $k_f \in \{5, 6, 7, 8, 9, 10\}$. Histograms reporting the frequency of improvements with

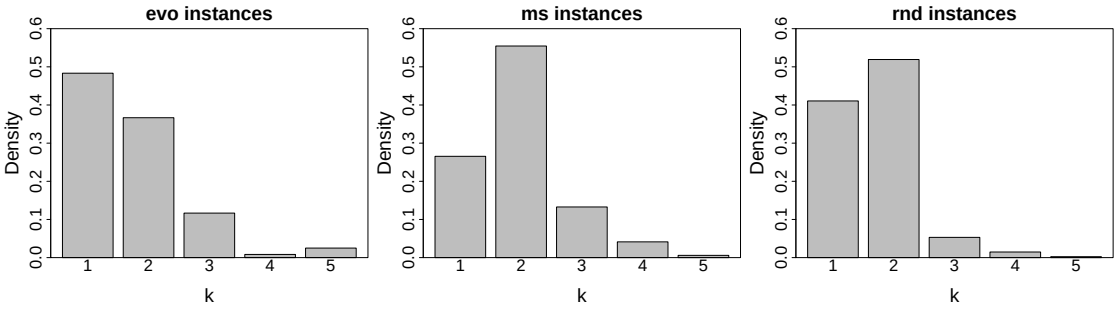


Figure 6.8: Histograms showing the fraction of times an improvement is found with neighbourhood of size k.

specific values of k for `LNS-1` are drawn in Figure 6.8. As it can be observed, most improvements are found for neighbourhood sizes of 1 and 2; however, a significant fraction of improvements is achieved with k equal to 3 and 4. Neighbourhoods corresponding to k up to size 5 were used by the algorithm. This analysis shows that neighbourhoods larger than 2 have to be considered in order to have a method with high performance. One may observe that for $k_f \leq 2$ the problem has a polynomial time worst case complexity and a dedicated polynomial time algorithm could be used instead of RecBlock. However, RecBlock is very fast also in this case because it exploits the constraints imposed by the fixed founder values to efficiently explore the search tree corresponding to the sub-problem restricted to assigning k founders. An *ad hoc* algorithm designed to solve efficiently the sub-problems with $k_f \leq 2$ could anyway make the overall technique more efficient.

Another insightful piece of information is given by the percentage of cases in which an improvement is achieved with a neighbourhood size smaller than that of the one for which the last improvement was achieved, i.e., after resetting k to k_{min} (see Line 12 in Algorithm 8). This analysis shows the utility of varying the neighbourhood size during the search. Table 6.5 reports the percentage of times such an event occurred for each instance class. The value p_{ij} in row i and column j means that an improvement with $k = j$ has been achieved $p_{ij}\%$ of times after an improvement attained with $k = i$. As we can note, a significant percentage of improvements has been achieved by moving to a smaller neighbourhood.

6.8.6 `LNS-1` speed-up

The algorithm `LNS-1` can be improved by adding a caching mechanism that makes it possible to avoid revisiting neighbourhoods included in previously explored ones. The possibility of revisiting (part of) a same neighbourhood arises when an improvement is found for a given set $\mathcal{I}_\mu$ of μ founder indices: in the next iteration k is reset to 1 and the search continues by exploring neighbourhoods of size 1. All solutions that can be obtained by exploring sequences of founders whose indices are in a set $\mathcal{I}' \subseteq \mathcal{I}_\mu$ are provably not better than the current best solution, because the neighbourhood exploration is complete. Therefore, computation time can be saved by avoiding exploring subsets of already con-

Table 6.5: Percentage of improvements achieved with a smaller neighbourhood w.r.t. the last improvement. The value p_{ij} in the matrix means that, after an improvement achieved with a neighbourhood i, a further improvement has been achieved with a neighbourhood of size $j < i$ in p_{ij}% of the cases.

`evo` instances			
	1	2	3
2	4.6	–	–
3	7.1	7.1	–
4	0	0	0
5	0	0	33.3

`ms` instances			
	1	2	3
2	6.9	–	–
3	0	24.4	–
4	0	21.4	7.1

`random` instances			
	1	2	3
2	23.7	–	–
3	1.5	46.1	–
4	11.1	5.6	0

sidered founders. Since the neighbourhoods are scanned in a random order, for every $k \leq \mu$ a subset of an already explored set of founders can be visited with probability $\binom{\mu}{k}/\binom{k_f}{k}$. We implemented a caching mechanism based on recording the index set $\mathcal{I}^*$ of the last visited founders that led to a solution improvement: as long as no improvement is found, before exploring a neighbourhood, we check whether it consists of founders with indices all included in $\mathcal{I}^*$. The set $\mathcal{I}^*$ is updated whenever a solution improvement is found.

We compared the performance of simple `LNS-1` against that equipped with the caching mechanism, named `LNS-1c`. The introduced variant only affects execution time, because it does not perturb the search process. Therefore, no difference whatsoever is expected—nor has it been found—with respect to solution quality. Results show that `LNS-1c` can be up to 15% faster than `LNS-1`. The average speed-up attained on `evo` instances is 2%, with a maximum of 6%. Conversely, for `ms` and `random` instances the improvement is more evident, amounting on average to 4% for both sets, with a maximum of 14% and 15%, respectively. Hence, `LNS-1c` is the variant we use for the comparison against the state of the art.

6.9 Comparison of `LNS-1c` against the state of the art

In this section, we compare `LNS-1c` with the state-of-the-art algorithms RecBlock (both complete and incomplete versions) and B&F IG. Tables 6.6, 6.7 and 6.8 report the results of `LNS-1c`, RecBlock-comp, RecBlock-incomp and B&F IG on the three benchmark sets.

Instances were solved with $k_f \in \{5, 6, 7, 8, 9, 10\}$. Therefore, the algorithms are compared on a set composed of 108 cases. The total time allotted to the methods was 3 days (including initial solution's computation time) for each run. The best solution value returned is highlighted in boldface; in case of ties, the shortest time is also marked.

6.9.1 Comparison with RecBlock

We compared LNS-1c with RecBlock with the latter both run as a complete (RecBlock-comp) and incomplete algorithm (RecBlock-incomp). In our experiments, we run the default incomplete variant of RecBlock. We chose not to include RecBlock-D0C1 version in our comparison because, as can be clearly seen in Tables 6.2 and 6.3, this variant returns poor solutions. RecBlock-D0C1 is meant to run fast, but in our comparisons we are more interested in high solution quality and time limit is not the limiting factor. Indeed, each run of the plain version of RecBlock is composed of two parts: in the first, the algorithm is run in an incomplete version and its solution is used as upper bound provided to the complete version. As observed in initial experiments, the default parameter setting is the one that enables RecBlock-incomp to attain a very good trade-off between solution quality and execution time. A careful parameter tuning could improve the performance of RecBlock-incomp on specific instances or restricted classes of instances; nevertheless, an automatic parameter tuning for RecBlock would require a high amount of computation time and the resulting algorithm would anyway not be guaranteed to find a solution in the allowed time limit. In fact, we would like to emphasise that the strength of RecBlock lies in its efficient tree exploration, rather than in its capabilities as heuristic algorithm.

Results from Tables 6.6, 6.7 and 6.8 show that many instances could not be solved by RecBlock-comp within the time limit of 3 CPU days. RecBlock-comp could solve only 33 out of 108 cases. Notably, in all the 33 cases, LNS-1c found the same solution value RecBlock-comp found, which is the optimal solution. Furthermore, also the performance in terms of computation time is strongly in favour of LNS-1c. Indeed, only in 6 cases out of 33 RecBlock-comp is faster than LNS-1c. The Wilcoxon test confirms this hypothesis.

RecBlock-incomp returned a feasible solution in 90 out of 108 cases; we can observe that the cases in which no solution was returned are the ones with 9 or 10 founders. We compared the solution quality returned by the algorithms on the restricted set of cases in which a feasible solution was returned by RecBlock-incomp. Aggregate results are shown as boxplots of relative solution value difference in Figure 6.9.

LNS-1c returns better results than RecBlock-incomp in all evo and ms instances, except for a single case in the ms set. In these test sets, LNS-1c returns significantly better solutions than RecBlock-incomp, with differences of up to 15%, and a median value of around 6%. Results on random instances are less clear, as LNS-1c returns better results in 18 out of 33 cases and has a lower median, but the Wilcoxon test does not reject the null hypothesis of equal performance.

A further comparison between LNS-1c and RecBlock-incomp can be done in terms of quality with respect to time, similarly to what is done with run time distributions [110]. Nevertheless, RecBlock-incomp does not return a series of solutions during time, but just one; in fact, it requires a finite time for returning one result at the end of the exploration of a heuristically pruned search tree and it does not exploit longer execution times. Therefore, run time distributions can not be computed. Instead, for each instance, we stored the results returned by LNS-1c in the time RecBlock-incomp found a feasible solution to the instance. Results are shown in Figure 6.10, while detailed results can be found

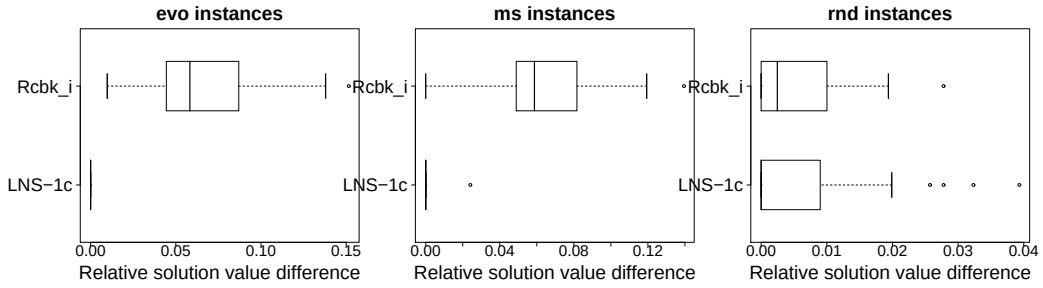


Figure 6.9: Boxplot of relative solution value difference between `LNS-1c` and RecBlock-incomp (labelled as Rcbk_i in the graphics).

in [183]. We can note that `LNS-1c` is superior to RecBlock-incomp on both `evo` and `ms` instances, with an average improvement of 4% and 5%, respectively. Conversely, on `random` instances RecBlock-incomp returns better solutions, with an average improvement of 3% over `LNS-1c`. The performance of `LNS-1c` on `random` instances could be improved by substituting RecBlock-comp with RecBlock-incomp in the neighbourhood exploration of `LNS-1c` and limiting the maximum number of neighbours visited (*maxCombinations* parameter). Thus, the time spent in exploring a neighbour is considerably reduced, especially when k_f is large, and more steps of local search can be performed. We implemented such a variant and the results show that the average gap to RecBlock-incomp is halved. However, it is important to remark that providing solutions to the FSRP in short execution times is not a requirement and, as from the results we have discussed before, `LNS-1c` is able to return better solutions on average by exploiting the whole computation time allotted. Moreover, the `evo` and `ms` instances are generated according to biological models and are realistic, in the sense that a real-world instance would very likely have a similar structure. On the contrary, `random` instances are a purely artificial testbed and are meaningless with respect to biological applications.

6.9.2 Comparison against Back-and-Forth Iterated Greedy

We also compared `LNS-1c` with B&F IG, that attained good results on large-size random instances, as shown in Section 6.6.2.

The timeout for B&F IG was set to 3 days for each case, as in the previous comparisons. As it can be seen from boxplots in Figure 6.11, `LNS-1c` attains considerably better results than B&F IG. `LNS-1c` finds better solutions than B&F IG most of the times: this concerns 25 cases from `evo` and 35 cases from both `ms` and `random`. In the few remaining cases, B&F IG returns results of the same quality as `LNS-1c`, except for one case in the `random` set, in which it finds a better solution—one unit lower than that returned by `LNS-1c`. These results

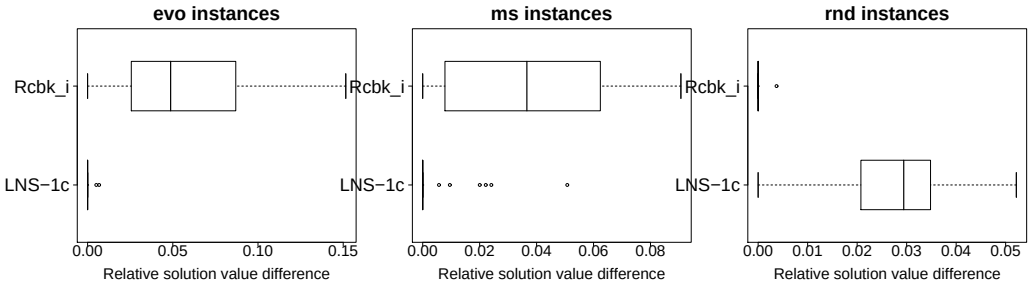


Figure 6.10: Boxplot of relative solution value difference between `LNS-1c` and RecBlock-incomp within the time required by RecBlock-incomp (labelled as Rcbk_i in the graphics).

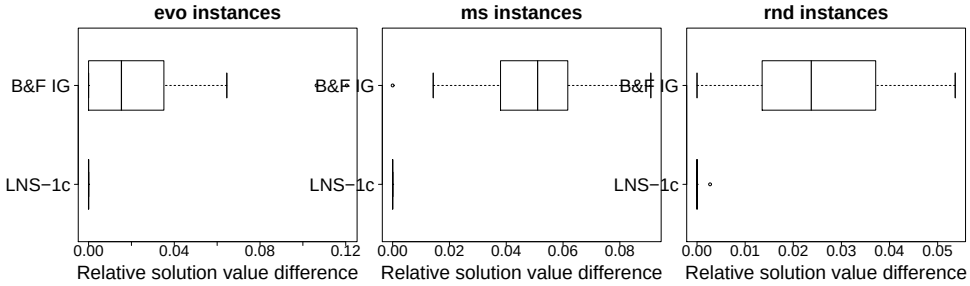


Figure 6.11: Boxplot of relative solution value difference between `LNS-1c` and Iterated Greedy.

show that `LNS-1c` can indeed considerably improve the initial solution provided by B&F IG and that it is far more efficient, because it can successfully exploit the available computation time.

6.10 Conclusions and discussion

In this chapter, we have described LNS algorithms we designed for tackling large-size instances of the FSRP. The algorithms exploit RecBlock, a state-of-the-art complete technique for this problem. The best algorithm, named `LNS-1c`, performs a local search in which a move consists in reassigning the values of one or more founders. Given a founder matrix, its neighbours of rank k are all the founder matrices differing in k founders. The exploration of such large neighbourhoods is performed by RecBlock, that returns the provably best founder configuration among all the ones in the neighbourhood. `LNS-1c`

achieves a performance considerably better than that of the current state-of-the-art methods. The algorithm also combines the property of being *anytime solvers* with that of completeness, if time is not limited. The experimental analysis has been performed on three sets of instances: `evo`, `ms` and `random`. `evo` and `ms` instances are created according to evolutionary models, so as to have realistic instances of variable size, whilst `random` instances have been considered just for comparison with previous works, being not meaningful from a biological perspective. Indeed, real-world instances are very likely to be far from random.

Further variants of LNS could be designed and embedded into higher level techniques, such as Iterated Local Search and Memetic Algorithms. Future work is also planned for tackling variants of the FSRP, such as the ones including mutation, noise and missing values. Furthermore, other variants including additional biological constraints, e.g., evenly distributed breakpoints or maximal number of breakpoints per recombinant, can be tackled.

Table 6.6: Detailed results of `LNS-1c`, RecBlock-comp, RecBlock-incomp and B&F IG on the `evo` instances.

number of	LNS-1c		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)
evo-30_60								
5	**145**	**4**	**145**	87	152	4	**145**	**4**
6	**94**	**53**	**94**	1205	97	28	**94**	96
7	**65**	**86**	**65**	14555	68	225	66	32
8	**45**	**353**	**45**	21288	47	1354	47	7
9	**36**	**51**	—	—	39	16704	**36**	200
10	**28**	**1**	—	—	—	—	**28**	**1**
evo-30_90								
5	**203**	**60**	**203**	153	210	6	204	3478
6	**118**	**52**	**118**	1508	128	39	**118**	2932
7	**69**	**19**	**69**	15484	75	293	**69**	**19**
8	**43**	**3**	**43**	4316	47	2285	**43**	**3**
9	**35**	**69**	**35**	33399	37	27359	37	3
10	**31**	28	—	—	—	—	**31**	**21**
evo-30_150								
5	**381**	893	**381**	**287**	392	16	388	44436
6	**230**	**72**	**230**	3066	244	65	234	88
7	**131**	**56**	**131**	23034	138	501	133	5067
8	**63**	**59**	**63**	104006	70	5650	64	566
9	**39**	**1**	**39**	43334	40	40989	**39**	**1**
10	**35**	12	—	—	—	—	**35**	**7**
evo-50_100								
5	**368**	**145**	**368**	448	394	13	371	143
6	**250**	**113**	**250**	9976	263	67	255	14419
7	**174**	**14706**	**174**	250975	198	357	180	28460
8	**124**	149	—	—	139	5084	132	102223
9	**99**	2507	—	—	114	36290	104	933
10	**83**	3696	—	—	—	—	84	17557
evo-50_150								
5	**522**	**132**	**522**	382	553	19	528	191564
6	**319**	**109**	**319**	10228	341	124	320	96282
7	**205**	**4**	**205**	88294	207	606	**205**	**4**
8	**135**	169	—	—	141	4695	138	12871
9	**101**	108	—	—	107	48300	102	77
10	**82**	291	—	—	—	—	85	201
evo-50_250								
5	**1126**	3060	**1126**	**1604**	1182	107	1160	9574
6	**726**	**1060**	**726**	25095	744	262	752	144651
7	**450**	259	—	—	466	1397	465	63669
8	**258**	603	—	—	291	14324	270	8746
9	**141**	12100	—	—	159	177159	156	3599
10	**83**	275	—	—	—	—	93	78290

number of	LNS-1c		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)

Table 6.7: Detailed results of LNS-1c, RecBlock-comp, RecBlock-incomp and B&F IG on the ms instances.

number of	LNS-1c		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)
ms-30_60								
5	**124**	**209**	**124**	5368	130	8	126	128
6	**100**	98859	—	—	105	57	104	176786
7	**81**	17273	—	—	85	409	83	16
8	**70**	54798	—	—	76	6591	71	93
9	**60**	2002	—	—	65	72924	61	2406
10	**50**	38579	—	—	—	—	**50**	**21576**
ms-30_90								
5	**167**	**747**	**167**	2095	175	8	175	98780
6	**136**	**768**	**136**	163768	140	60	141	30
7	**114**	30934	—	—	119	570	117	225805
8	**97**	126402	—	—	102	8495	102	117982
9	85	216	—	—	**83**	226474	88	75
10	**74**	1648	—	—	—	—	77	6520
ms-30_150								
5	**251**	4986	**251**	**1805**	271	14	270	1412
6	**189**	**1421**	**189**	90048	204	93	203	43967
7	**153**	25361	—	—	159	1481	159	27490
8	**125**	7590	—	—	135	11381	133	8723
9	**103**	106022	—	—	—	—	109	116217
10	**88**	22794	—	—	—	—	92	33269
ms-50_100								
5	**310**	**2192**	**310**	166497	347	14	323	20093
6	**251**	18039	—	—	286	150	266	146944
7	**212**	442	—	—	220	1572	223	1739
8	**178**	51495	—	—	188	13538	187	2585
9	**155**	38758	—	—	163	218293	163	104530
10	**137**	30080	—	—	—	—	141	14612
ms-50_150								
5	**429**	48449	**429**	**27077**	453	35	468	2177
6	**346**	26957	—	—	376	114	368	24719
7	**286**	1958	—	—	312	1049	300	110129
8	**241**	130741	—	—	257	12294	254	11687
9	**203**	170493	—	—	—	—	212	35375
10	**174**	8253	—	—	—	—	183	250919
ms-50_250								
5	**613**	**2171**	**613**	18235	649	84	646	251801
6	**479**	48013	—	—	516	364	515	6793
7	**396**	16430	—	—	433	6383	418	212274
8	**336**	23916	—	—	360	28984	363	15497
9	**283**	243608	—	—	—	—	306	250687
10	**248**	7413	—	—	—	—	268	1906

Table 6.8: Detailed results of LNS-1c, RecBlock-comp, RecBlock-incomp and B&F IG on the random instances.

number of	LNS-1c		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)
random-30_60								
5	**372**	48427	**372**	**38490**	376	5	376	637
6	**324**	44255	—	—	333	30	327	214250
7	**293**	906	—	—	295	232	294	179134
8	**268**	96096	—	—	270	1482	270	185210
9	246	175659	—	—	**245**	7502	249	120582
10	229	190559	—	—	**225**	181678	230	2605

number of founders	LNS-1c		RecBlock-comp		RecBlock-incomp		B&F IG	
	value	time (s)	value	time (s)	value	time (s)	value	time (s)
random-30_90								
5	**585**	72903	**585**	**58301**	594	7	595	177509
6	**516**	179754	—	—	526	63	527	24720
7	472	55418	—	—	**469**	383	477	184758
8	**426**	107173	—	—	**426**	**2437**	441	92253
9	399	12679	—	—	**389**	13334	406	174833
10	370	244167	—	—	**356**	108248	369	55441
random-30_150								
5	**976**	**134777**	**976**	169057	986	24	1000	35683
6	**865**	216875	—	—	876	106	890	8264
7	**778**	140918	—	—	781	499	812	96348
8	**710**	250463	—	—	713	4044	736	7448
9	666	87405	—	—	**648**	35617	685	76775
10	**619**	21046	—	—	—	—	639	136143
random-50_100								
5	**1213**	65968	—	—	1226	18	1226	47222
6	**1097**	60881	—	—	1108	63	1115	164682
7	1009	8769	—	—	**1000**	405	1024	31644
8	928	44145	—	—	**927**	2836	948	164238
9	875	113792	—	—	**862**	17875	892	109557
10	830	221118	—	—	**804**	167759	849	81393
random-50_150								
5	**1800**	195873	—	—	1815	23	1838	191857
6	**1622**	144474	—	—	1626	138	1673	59084
7	**1484**	221180	—	—	1490	783	1544	148594
8	1385	85140	—	—	**1381**	3353	1446	92349
9	1302	222181	—	—	**1280**	26654	1351	256516
10	**1240**	244166	—	—	—	—	1285	196005
random-50_250								
5	**3043**	101246	—	—	3074	117	3126	102337
6	**2725**	172785	—	—	2751	290	2845	110523
7	**2508**	251951	—	—	2512	1352	2628	119105
8	2330	176486	—	—	**2320**	5698	2455	104077
9	2204	244380	—	—	**2161**	46765	2318	180631
10	**2097**	257557	—	—	—	—	2194	205293

Part II

Boolean Network Design

Chapter 7

Brief Introduction to Boolean Networks

In this chapter we formally introduce Boolean Networks (BNs), a model born as a mathematical tool to study Gene Regulatory Networks (GRNs) and complex systems in general, which lately garnered much attention. The literature on BNs is vast and branches into statistical physics, dynamical system theory, chaos theory, control theory, information theory and also bioinformatics. Lately, BNs have also been subject of investigation as to their applicability as robot controllers [34].

This chapter offers a brief overview of the concepts relevant to this thesis and will point out basic definitions and terminology; finally in Section 7.2, we introduce and motivate the use of BN and metaheuristics in the context of the ensemble approach, which studies GRN models from the perspective of complex system biology. Our aim is to eventually formulate a methodology to automatically build models of dynamical (complex) systems which satisfy certain constraints and objectives. These constraints and objectives allow us to define a combinatorial optimisation problem which will be tackled by metaheuristic algorithms.

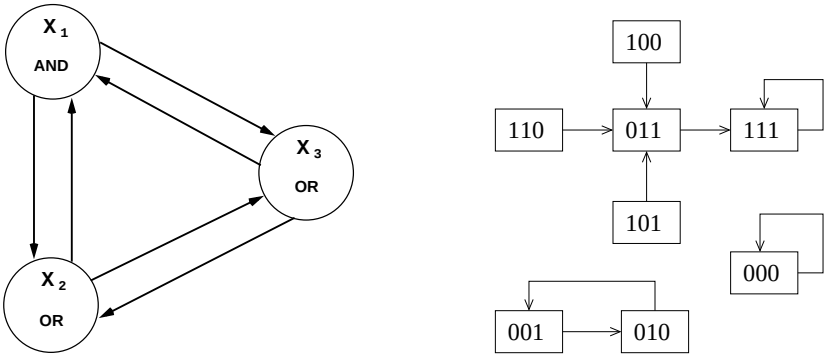
7.1 Boolean networks

BNs have been firstly introduced by Kauffman [125, 127] and subsequently received considerable attention in the composite community of complex systems. Recent advances in this research field can be mainly found in works addressing themes in GRNs or investigating properties of BNs themselves [4, 5, 80, 179, 201].

A BN is a discrete-state and discrete-time dynamical system whose structure is defined by a directed graph of N node[1], each associated to a Boolean variable $x_i \in \{0, 1\}$, $i = 1, \ldots, N$, and a Boolean function $f_i(x_{i_1}, \ldots, x_{i_{K_i}})$, called node transition function[2], where K_i is the number of inputs of node i. The arguments of the Boolean function f_i are the values of the nodes whose outgoing edges are

[1] In this part, we prefer the terminology "node" as opposed to "vertex" because it is more common in this area of research.

[2] For the sake of brevity, we will often omit "transition".



(a) A BN with three nodes (b) State space

Figure 7.1: An example of a BN with three nodes 7.1a and its corresponding state space under synchronous and deterministic update 7.1b. The network has three attractors: two fixed points, $(0,0,0)$ and $(1,1,1)$, and a cycle of period 2, $\{(0,0,1),(0,1,0)\}$.

connected to node i (see Figure 7.1a). Notice that, contrarily to what happens for graph, edges going into a node are *ordered*. The state $s \in \{0,1\}^N$ of the system at time t, $t \in \mathbb{N}$, is defined by the vector of the N Boolean variable values at time t, therefore, $s(t) \equiv (x_1(t), \ldots, x_N(t))$.

The most studied BN models are characterised by having a *synchronous* dynamics—i.e., nodes update their states at the same instant in parallel—and *deterministic* functions (see Figure 7.1b). In this thesis we consider only this kind of networks (which we simply call "Boolean networks" omitting synchronous and deterministic). However, many variants exist, including asynchronous and probabilistic update rules [88, 208].

State and configuration spaces. BN models' dynamics can be studied by means of usual dynamical systems methods [11, 204], hence the usage of concepts such as *state (or phase) space*, *trajectories*, *attractors* and *basins of attraction*. Note that we make a distinction between "configuration space" and "state space" notions. The configuration space is the set of states the network can assume, namely, $\{0,1\}^N$; in this space any kind of distance could be defined, but, in practice, the notion of Hamming distance is employed. Under Hamming distance, we can say that two states are neighbors, in the configuration space, if their distance is 1; this entails that the configuration space is an hypercube. On the contrary, the state space of a BN is, in general, a *directed* graph (an example is depicted in Figure 7.1b), therefore the state space notion emphasises the dynamic aspect of a network. In addition, the neighborhood definition is totally different between the spaces: in the configuration space we have Hamming neighborhood so, for instance, state vectors such 000 and 001 are neighbors; in the state space of the network in Figure 7.1a, such states are not neighbors[3]. The fact that BN dynamics is synchronous and deterministic has important repercussions on the topology of the state space. Specifically, every vertex in the state space graph has exactly one successor, or, in other words, out-degree

[3]They belong, in fact to two different basins of attractions (see below).

equal to 1; on the other hand, in-degree is not constant and can be as low as 0 in case of *garden-of-Eden states*.

Trajectories and attractors. Given this setting, a BN is a discrete map $F : \{0,1\}^N \to \{0,1\}^N$ and $s_{t+1} = F(s_t)$. Network trajectories in the N-dimensional configuration space are sequences of states $s, F(s), F^2(s), \ldots$. By determinism, we have that trajectories do not cross. Since $\{0,1\}^N$ is finite and the dynamics is deterministic, every trajectory eventually enters a (possibly degenerate) cycle. In other terms, every trajectory is composed of a *transient*, possibly empty, followed by an *attractor*, that is a cycle of period, or length, $l \in \{1, \ldots, 2^N\}$. Attractors of period 1 are also known as *fixed points*. The set of attractors (i.e., cycles in the state space graph) is called *attractor landscape*. Another important related concept is that of *basins of attraction*. The set of states B_i from which an attractor A_i can be reached is called basin of attraction of A_i. We also define the quantity $w_i = \frac{|B_i|}{2^N}$ called *relative basin weight*, and will be used in the remainder of the thesis. Basins of attraction of synchronous and deterministic BNs partition the configuration space.

Another topological property of the state space that descends from BN determinism is that every attractors state is the root of a directed tree whose branches are transients and whose leaves are garden of Eden states.

7.1.1 Random Boolean Network model

A special category of BNs that has received particular attention is that of RBNs, which can capture relevant phenomena in genetic and cellular mechanisms and complex systems in general. Recent advances in this research field, along with efficient mathematical and experimental methods and tools for analysing BN dynamics, can be mainly found in works addressing issues in GRNs or investigating properties of BN models [4, 80, 179, 201]. For instance, attractors of RBNs have assumed a notable relevance because they can be interpreted as cellular types [113] in GRN models. This interpretation has recently been extended by considering sets of attractors, the so-called *Threshold Ergodic Sets* (TES_θ), instead of single attractors [198, 228]. This extension provides support to the usefulness of RBNs as GRN models, as it makes it possible also to model cell differentiation dynamics.

RBNs are generated by choosing at random K inputs per node and by defining the Boolean functions by assigning to each entry of the truth tables a 1 with probability p and a 0 with probability $1 - p$. Parameter p is called *homogeneity* or *bias*. Depending on the values of K and p the dynamics of RBNs is called either *ordered* or *chaotic*. This distinction is based essentially on the stability of the dynamical attractors with respect to small perturbations: in the case of ordered systems the majority of nodes in the attractor is frozen and small perturbations usually die out, whilst in disordered ones attractor cycles are very long and the system is extremely sensitive to small perturbations, that is, slightly different initial conditions lead to divergent trajectories in the state space. To put it other words, ordered RBN systems usually have fairly regular basins on attraction, so that two nearby states[4] often evolve to the same attractor, while in disordered systems they often go to different attractors. This behaviour is

[4] According to the Hamming distance.

reminiscent of the "butterfly effect" and this provides a reason why disordered RBNs are often called "chaotic" (in spite of the fact that, since the attractors are cycles, the term pseudo-chaotic would be more appropriate).

RBNs temporal evolution undergo a second order phase transition between order and chaos, governed by the following relation between K and p:

$$K_c = [2p_c(1-p_c)]^{-1}$$

where the subscript c denotes the *critical* values [59]. Networks along the critical line show equilibrium between robustness and adaptiveness [4]; due to this property, RBNs are supposed to be plausible models of the living systems organisation on the basis of heuristic arguments which can be summarised as follows. Biological systems need a certain level of stability, in order not to be disrupted by fluctuations which can take place either in the system or in the environment, and they need at the same time to provide flexible responses to changes in the environment. While a chaotic system would be poor at satisfying the first need, a system deeply in the ordered region would fail to meet the second requirement. Recent results support the view that biological GRNs operate close to the critical region [9, 201, 211].

An important concept in system dynamics is that of Lyapunov exponent. For dynamical systems, the Lyapunov exponent characterises the rate of separation of two trajectories starting from two initial condition, s_0 and $s\prime_0$, distant, under some distance definition, an infinitesimal δ from each other. For a dynamical system with Lyapunov exponent λ we have that, in the limit $t \to \infty$, the distance between two trajectories grows as $e^{\lambda t}\delta$. It is known that, for RBNs, the Lyapunov exponent λ can be analytically calculated as follows: $\lambda = \log \xi = \log[2p(1-p)K]$ (equation 27 in [145]). If we substitute a bias p in the previous equation, we obtain networks in chaotic ($\lambda > 0$), critical ($\lambda = 0$) and ordered ($\lambda < 0$) regimes, respectively.

A useful related notion to analyse the dynamics of RBNs is the (average) network sensitivity, calculated by taking the average of all node function average sensitivities [129]. The average sensitivity of a Boolean function $f(x)$, where x is a Boolean vector, is the average number of 1-Hamming neighbours x' of x—that is, x and x' differ in only one position—such that $f(x') \neq f(x)$. More formally, the sensitivity s_f of a K-variable Boolean function f measures how sensitive f is to a change in its inputs, and is calculated as follows. Let us define $s_f(x) = |\{x\prime \mid f(x\prime) \neq f(x) \wedge d_H(x, x\prime) = 1\}|$ where $x \in \{0,1\}^K$ and d_H is the Hamming distance. The sensitivity is thus $s_f = \frac{1}{2^K} \sum_{x \in \{0,1\}^K} s_f(x)$. It has been shown that network sensitivity is closely related to the notion of Lyapunov exponent [210]. This is an important measure because it allows one to determine the dynamical regime of a RBN just by looking at its functions.

7.2 Motivations of designing by metaheuristics

The interest of BNs as GRN models relies primarily in the fact that some classes of BNs statistically reproduce some characteristics of real cells. For example, it has been shown that single gene knock-out experiments can be simulated in RBNs [203]. Reproducing statistical properties of real cells through specific classes of GRNs is a *complex systems biology* approach [123]. A specific research

stream in this area is the *ensemble approach* [128], that aims at finding classes of GRN models which match statistical features of genes, such as the number of cell types of an organism or cell dynamics in case of perturbation. Currently, modern biotechnology tools, such as DNA microarrays, make it possible to gather a huge amount of biological data, therefore the approach of complex systems biology can be applied even more effectively than in the past.

Our long-term research goal is to develop tools and methodology for automatically designing GRNs meeting given requirements. This methodology can make it possible to address the problem of designing GRNs with specific dynamic behaviour (e.g., problems in reverse engineering GRNs) and to achieve advancements in specific research lines following the ensemble approach and complex systems biology in general.

A first proposal of such method is detailed in Section 9.1.2 of which we give now only a brief overview. Our methodology consists of transforming the design problem into an optimisation problem. Suppose that we want to study a class of systems with the ensemble approach and also suppose that systems in such class have a certain set of statistical features in some feature space $\mathcal{F}$. What we want is to search in the space of BNs for an ensemble of networks that have features similar, i.e., close according to a distance measure in $\mathcal{F}$, to the one characterising the system studied. This similarity condition provides us with an objective function. Our combinatorial formulation can be further enriched by adding constraints, i.e., additional characteristics that all BNs should have, which can come from heuristic knowledge. As an abstract example, we could ask to limit our search to BNs whose node functions satisfy some conditions. After having defined a (constrained) combinatorial optimisation problem, we can then apply a generic metaheuristic search. In our current model, the metaheuristic searches in the Boolean function space, but other options are, of course, possible. We are also not limited to one particular metaheuristic: in Chapter 9 we review case studies that employ both local searches and population methods.

An important aspect of this methodology is worth noticing. Defining an objective function entails performing one or more simulations of a network and gathering statistical data. This has two important repercussions on the practical side:

1. repeated simulations of a network might be extremely computationally expensive. For instance, suppose one wants networks possessing a peculiar attractor structure: that necessitate the sampling of the attractor landscape. Suppose now that the search encounters a chaotic network, characterised, as we know, by long attractors; but the longer the attractor the longer the simulation lasts. In the end, the evaluation of chaotic network for this specific task might be prohibitive.

2. Since BNs are complex systems, it is difficult or plainly impossible to implement a speed-up technique typical of local search algorithms, that is, delta evaluation. Usually, a move changes only a small part of a solution and in many problems (for instance, the Travelling Salesman Problem) it is possible to calculate the value of a neighboring solution by adding a delta (positive or negative) to the value of the current solution. Given the non-linear dependencies between network nodes, it is difficult to predict the effect on the objective function of a small change: every new solution (i.e., BN) encountered, thus, has to be simulated.

In our opinion the application of metaheuristics to this topic can be fruitful for the following reasons:

- the object to model is typically a complex system: the relationships between its components are often difficult to model directly or simply unknown, and a reductionist approach alone is not able to cope with complexity. As we wrote in Chapter 1, the upper-level algorithm that characterises the architecture of metaheuristics is, in fact, a learning method that guides the underlying search. This advantage can be exploited to effectively explore the search space because the algorithm is able to "learn" the relations between components of a complex system.

- Metaheuristics are a flexible framework which can be extended in multiple ways: one can, for example, integrate problem specific knowledge or hybridise several techniques. Integration is not limited to search algorithms alone: we believe that machine learning methods can also prove useful.

- Metaheuristic trade optimality for efficiency. As also written at the end of Section 2.2.1 and restated in Section 9.1.2, we do not aim to solve a problem to optimality because the objective function is inherently inaccurate and has more the role of a guide to the search process.

As we will discuss in the following chapter, application of metaheuristics to the design problems requires us to have appropriate software tools, for both network simulation and analysis.

7.3 Conclusions and discussion

In this section we introduced BNs as computational models to study system biology; we gave an overview of the terminology and illustrated the main concepts used throughout this thesis. In Section 7.1.1, we discussed in more detail Random Boolean Networks, computational objects that proved useful to modelers to describe certain biological organisms and phenomena. We concluded with Section 7.2, where we introduced the ensemble approach as the main motivation of our automatic design framework by metaheuristics, and we anticipated elements of the design methodology itself, to be fully described in Section 9.1.2.

Chapter 8

The Boolean Network Toolkit

In this chapter we describe our effort to design and implement a flexible and efficient Boolean network simulator. This simulator, called Boolean Network Toolkit, is the software we extensively used to realise all our simulations and analyses reported in Chapter 9. Boolean Network Toolkit is open source and freely available [16].

8.1 Introduction and Motivations

In this section we discuss the desiderata of a hypothetical BN simulator and we give a brief overview of available tools in the research community.

Leveraging a stochastic search to the automatic design of BNs entails that such an algorithm has to explore an enormous search space. In the simplest and most straightforward application, this search space is populated with BNs. Of course we could define a search space whose elements are models of BNs; for instance, we could establish that a point in the search space is a tuple $\langle N, k, p \rangle$ that completely defines a family of Random Boolean Networks. In a sense, this is a typical choice in the domain of Genetic Algorithms, where there can some degree of indirection between the actual solution of a problem (the *phenotype*) and its representation (the *genotype*). If we limit ourselves, as we will actually do in the case studies examined in Chapter 9, to the most direct representation that maps a point in the search space to a single BN instance, we have an exponentially large space. For example, a space inhabited by networks with $N = 20$ nodes and *fixed topology* with constant in-degree $K = 2$, would contain 16^{20} networks, that is, all the possible ways of assigning one of the Boolean functions of 2 inputs (in total 2^{2^K}, hence 16) to each of the 20 nodes. Oftentimes we lack an effective heuristic to explore the search space, with respect to some objective function, or, equivalently, a prior probability distribution that could help the search process. As a counterexample, we know that if we hypothetically searched for networks with very long transients, we could sample the space of chaotic RBN which are likely to have such a feature (see Section 7.1.1).

It appears clear that, in order to explore such a large search space effectively, we both need sophisticated search algorithms and an efficient simulator. Indeed

our search method will likely have to sample the search space many times and each of these samples, a BN, will have to be evaluated, i.e., simulated, and, likely, the simulation will be the most computationally expensive operation. As a matter of fact, this is the case in the applications studied in Chapter 9.

Efficiency is not only the main concern of our simulator. From a practical point of view, it is hard and time consuming to design and program good metaheuristic algorithm. For this reason, third party libraries are very valuable to the practitioner; for example, in Chapter 4 we mentioned that many GA libraries exist and we made use ourselves of the EasyLocal++ framework in Chapters 3 and 4. Being able to integrate our simulator with metaheuristic software libraries is thus a major prerequisite.

Other important aspects are that of modularity and flexibility. Our simulator should be flexible enough to allow the researcher to quickly set up experiments of different kind which go beyond the usual basic tasks, like computing a BN's trajectory or finding a set of attractors. For instance, the experiment described in [198] to find threshold ergodic sets is rather complex and requires a high degree of customisability; our simulator should provide all that.

As a further prerequisite, our simulator should not also be limited to synchronous BNs: our methodology (fully described in Section 9.1.2) is independent of the underlying network model and several different update schemes are available that might be worth investigating (some of which are reported in Section 7.1). Moreover, in a future, we might need to experiment with models alternative to ordinary BNs. It is natural to think that, a well-engineered simulator is capable of handling not only deterministic BNs, but also other slightly different models of networks: indeed this is a prerequisite of our hypothetical simulator, albeit a minor one. We do not aim, though, to a too generic piece of software: our simulator's main area of applicability should be that of discrete-time network models for biology, typically genetic regulatory networks. We could in principle simulate other network models, like Artificial Neural Networks (ANN), but that would be beyond the scope of the simulator, and, in particular, many *ad hoc* tools for ANN are available, which are both efficient and feature-rich.

After this informal discussion, we can now summarise our requirements for such a simulator software that will be the driving design principles of the Boolean Network Toolkit, as shown in Section 8.2.

Flexibility. The simulator should allow the researcher to implement different kinds of experiments in diverse contexts and under different constraints. It should also provide a way to easily modify a BN, a fundamental operation required by local search metaheuristics.

Ease of integration. The simulator is going to be integrated with other software libraries, typically written in `C++` for greater efficiency.

Free and Open Source Software. We strive to maintain our software free so as to foster adoption and extension.

Modularity and separation of concerns. Should be able to support different BN update schemes and possibly different network models. Also, modularity is a key to achieve flexibility by allowing the researcher to modify and extend the tool itself.

Efficiency. The simulator should be as efficient as possible.

8.1.1 Available tools

In this section a brief overview of available simulation tools for BN models is given.

There are a number of software applications for experimenting with BNs. Unfortunately, none of them fully satisfies the stated prerequisites. Some of them are too narrow in scope, inefficient or difficult to integrate. Many tools also come with an extensive graphical user interface which, although beneficial in some cases, is not really needed in our specific application. Often, the GUI is the only way to interact with the simulator while, on the contrary, we want to be able to run our simulators off-line. Good design principles (like MVC) promote separation of concern so that application business logic and presentation are clearly separated and, thus, loosely coupled; for this reason, Boolean Network Toolkit does not come with a GUI environment, which can, of course, be added at a later stage without changing the core of the simulator.

There are two pieces of software written in Python: one is `BNSim` [33], a fairly simple BN simulator rather limited in scope, the other is `BooleanNet` [3, 120] which, on the contrary, supports several update schemes including piece wise differential equation updates. Two libraries roughly equivalent to `BooleanNet` are `Random Boolean Network Toolbox` [196], a library written in MATLAB, and `BoolNet` [159, 160], an `R` package. `Boolean Network Modeler` [63] is an application written in `C#` that provides a graphical interface to design and analyse BNs. Another interactive graphical tool is `NetBuilder` [235], a piece of software, written in Microsoft Visual `C++`, capable of simulating networks driven by several update functions, such as Boolean functions but also differential equations. A Python version, called `NetBuilder'` is also available [194]. Another interactive graphics software for researching discrete dynamical networks is `DDLab` [241, 242]; this application is oriented towards the study of Boolean models and offers an extensive array of analytical tool and visualisation diagrams. One last simulator worth of notice is `RBNLab` [89], written in Java, that simulates RBNs under several update schemes.

8.2 Software Design

This section briefly explains the design choices made to implement the Boolean Network Toolkit. First we will introduce the key abstractions in the simulator pertinent to BNs. Afterwards, we describe the architecture of the simulator and we will see how these abstractions can be extended to encompass different kinds of update schemes and possibly new network models.

The requirements listed at the end of Section 8.1 push us to choose `C++` as our implementation language; the main programming paradigm adopted is, therefore, Object Oriented, but, as we will see, we will also employ elements of functional programming.

In this part we also show `C++` code to exemplify some important concepts; we have to warn the reader that such code does not exactly correspond to what is available in the simulator: names may differ and low-level details are removed, but the overall "feeling" is maintained.

8.2.1 Fundamental abstractions

The main abstraction to design is that of synchronous BN. In the Object Oriented paradigm, it would be intuitive to model a BN as a mutable object composed of a state vector and a topology. On the other hand, such abstraction is rather distant to the original mathematical concept that sees BNs as discrete maps, i.e., functions. In formal term (see Chapter 7), a BN is a function F that maps the current state to the next state $s_{t+1} = F(s_t)$. Notice that the form of F is neither dependent on the state nor the time parameter. Moreover, from a computational point of view, a function is an immutable object. The functional programming paradigm is the preferred approach to model BNs in the Boolean Network Toolkit.

The most basic simulation task is to evolve a BN a certain number m of steps starting from initial condition s_0. With a functional approach this is straightforward. Formally we just need to take the first m elements from the sequence $F^i(s_0), i \in \mathbb{N}$; what we need is a way to *lazily*[1] represent such sequence, or, in computer science parlance, a *stream*. Lazily evaluated sequences, such as lists, are typically available in all major functional programming languages, such as Scheme, Haskell, but also Python and many others. `C++` does not directly support streams, but they can be implemented by means of objects (see, for instance, the Iterator Pattern in GoF [85] for a similar concept and iterator implementations in all mainstream OO programming languages, such as Java and C#; in particular, see also C# generators [237] and the `yield` keyword).

Almost all computational tasks on BNs involve computing a trajectory, therefore the concept of stream, or lazy list, is pervasive in the simulator. Such abstraction is fundamental because, as we will see, it allows us to easily decompose complex experiments on BNs into simpler and reusable components.

With a trajectory, it is easy to find an attractor: it is sufficient to apply one of the well-known cycle finding algorithms. In the Boolean Network Toolkit we implemented two algorithms. One is a naïve search that memorises every state encountered and looks for repeated values. This algorithm does not scale well, in terms of both time and space complexity, with long transients, therefore we also provide an alternative, Brent's cycle detection algorithm [36] (see also [133, 236]), which scales much better for long transients (chaotic RBNs, for instance), but is slower for short sequences. Finding an attractor starting from an initial condition can be in turn interpreted as a higher-order function which takes a BN (a function in our interpretation), an initial state and returns an attractor object. Let us write the type of such function as $State \to Attractor$.

Now that we have encapsulated the computation of an attractor into a function, we can easily calculate a set of attractors starting from a sequence of initial conditions. To do so we can use another important higher-order function on lists called `map`, that, roughly, is an abstraction of a 'for' cycle. `map` is a function that takes a function from a type A to another type B ($A \to B$ for short), a list of element of type A ($[A]$ for short) and returns a list of elements of type B (in symbols `map` : $(A \to B), [A] \to [B]$). If we combine a cycle detection strategy with this `map` abstraction, it is easy to obtain a function that takes a stream of states and returns a stream of attractors.[2] If we partially apply `map` to a cycle detector, such as a hypothetical `brentAlgorithm`, we have a function

[1] Lazy roughly means "computed on demand".

[2] Remember that in the functional world, lists are lazy.

$[State] \to [Attractor]$ from (stream of) states to (stream of) attractors.

Implementation of streams. The stream abstraction is encapsulated in `C++` objects, which, in `C++` terminology, are called *ranges*, fundamental concepts in the STL [205]. Ranges expose the same interface as commonly used STL containers, such as `std::vector`s and `std::set`s, namely, they provide methods to get a pair of iterators, one pointing to the beginning of the range (method `begin()`) and the other pointing to the end (method `end()`). Of course, in case of infinite ranges, like, for instance trajectories of BNs, the end iterator is simply a dummy object and does not point to anything meaningful. The implementation choices in the Boolean Network Toolkit make the aforementioned abstraction of stream interoperable with a large variety of algorithms already available in `C++` and, above all, the STL itself. For example, it is easy to print to standard output a range of attractors using familiar STL idioms, provided that an attractor is a printable object, as actually is in the Boolean Network Toolkit.

Drawing inspiration by the Boost Range Library [35], ranges can be easily composed together with small effort and convenient syntax. Every range defines, in fact, an operator '`|`' (pipe) which takes as first operand a range and as second operand a function; the semantics of pipe is the same as the `map` function.[3]

8.2.2 Simulator Architecture

In this section we give an overview of the simulator structure.

The main entity in the simulator is, of course, the BN. The principal design choice that can help us to achieve good modularity and flexibility is illustrated in the following. In the Boolean Network Toolkit network state, dynamics and topology are separate concepts. Network state can be any indexable data structure whose elements are Booleans, but in principle also integers or double precision floats depending on the domain of node values; this is a crucial characteristic if we want to extend the simulator with other network models. For BNs we employ the compact bitset structure provided by the Boost Dynamic Bitset Library.

Network topology is a stateless entity, structurally represented by a directed graph data structure whose nodes, indexed by integers, containing their update function. A network topology exposes methods to access its graph structure and, most importantly, it provides a method to compute the next value of a node given the current state vector. In order to implement network topology, we use the excellent Boost Graph Library [212].

Network dynamics is an entity that realises a particular node update scheme and is, of course, coupled to network topology. Objects that realise node update algorithms for Boolean networks implement the interface `BooleanDynamics`, whose definition is given in Listing 8.1. It can be noted that the definition of `BooleanDynamics` is equivalent to the interpretation of BN we gave in Section 8.2.1, namely, an object that takes a state vector and returns another state vector.

A use-case example of this architecture is given below:

[3] More precisely, the correct `C++` type for the second operand would be function pointer or functor.

```
1  class BooleanDynamics {
2    virtual State update(State s) = 0; // a function State -> State
3  }
```

Figure 8.1: `BooleanDynamics` interface definition (extract).

```
1  BooleanNetwork bn = // read BN from file
2  BooleanDynamics* dyn = synchronous_dynamics(bn); // get an
       updater
3  State s0 = // an initial state vector
4  State s = (*dyn)(s0); // state at t=1
5  s = (*dyn)(s); // state at t=2
6  s = (*dyn)(s); // state at t=3
7  BrentCycleFinder finder(dyn);
```

where `finder` is a function object, implementing Brent's cycle detection algorithm, that takes a state vector and returns an attractor. Notice that neither `BooleanNetwork` or `BooleanDynamics` are stateful objects and that network state is externally maintained. This way, we are able to run several network evolutions in parallel without copying the same topology over and over.

Following good programming practices, a simulator user is encouraged to encapsulate its code into generic and reusable abstractions, such as classes or functions. Whenever a researcher writes an experiment in the Boolean Network Toolkit, he is advised to extend the simulator and to make its code available to others; this is one of the motivations why the simulator is open source software. For instance, all use cases illustrated in Section 8.3, are actually snippets taken from the source code itself, stripped of minor details, which have been encapsulated into generic functions for easy reuse.

8.3 Use Cases

In this section we show some use cases that demonstrate the programming style of the Boolean Network Toolkit.

Attractor set. Let us start with finding a set of attractors in an N node network starting from m random initial conditions. Implementation is in the following snippet:

```
1  RandomStateGen states(N, m);
2  BooleanNetwork bn = // read BN from file
3  BooleanDynamics* dyn = synchronous_dynamics(bn); // get an
       updater
4  NaiveFinder finder(dyn);
5  AttractorRange ar = states | finder;
6  std::set<Attractor> attractors(ar.begin(), ar.end());
```

where `RandomStateGen` is a range that yields m random Boolean vectors and `NaiveFinder` is a function object that implements the naïve cycle detection algorithm. In Line 5 we show how new ranges can be constructed by using the pipe operator and, in Line 6, we demonstrate how ranges interoperate with STL classes. We also notice that `Attractor` objects provide a comparison operator, so they can be inserted into sets.

We can also print the set of attractors using standard STL facilities:

```
1  std::copy(ar.begin(), ar.end(), // for all elements in range
2    std::ostream_iterator<Attractor>(std::cout,  // print them to
         stdout
3      "\n") // separated by a newline
4  );
```

Basins of attraction. A small extension to the previous use case is to enumerate the whole configuration space of a network and count the multiplicity of each attractor[4], so that we obtain their basin size. To do so, we use a `std::map` to count the occurrences and a `StateEnumerator` object, a range, like `RandomStateGen`, which yields all state vectors of a certain dimension N.

```
1  StateEnumerator allStates(N);
2  BooleanNetwork bn = // read BN from file
3  BooleanDynamics* dyn = synchronous_dynamics(bn); // get an
       updater
4  NaiveFinder finder(dyn);
5  AttractorRange ar = allStates | finder;
6  std::map<Attractor,int> m;
7  for(AttractorRange::iterator it = ar.begin(); it != ar.end(); ++
       it) {
8    if(m.count(*it) == 0) // new attractor
9      m[*it] = 1; // also inserts attractor into m
10   else
11     m[*it] += 1; // increment occurrences
12 }
```

where the `C++` programmer can recognise the familiar STL iteration idiom. We also recall that in the first case of the if statement (Line 9), the attractor pointed by iterator `it` is also inserted into the map. In the end, map `m` contains, for each attractor, the number of states in its basin.

If we make use of the `Counter` class provided by the toolkit, we can further shorten our code. `Counter` is a data structure, halfway between a set and a map, that serves the same purpose as our map `m` in the snippet above: it keeps a counter of all objects inserted, attractors in our case, but, in fact, it stores a unique copy of each object. A `Counter` can be constructed with a range object: the semantics is to scan the whole range and insert into the newly created `Counter` all objects in the range. The resulting code is the following:

```
1  StateEnumerator allStates(N);
2  BooleanNetwork bn = // read BN from file
3  BooleanDynamics* dyn = synchronous_dynamics(bn); // obtain an
       updater
```

[4]Of course, such experiment is feasible only for small networks.

```
4   NaiveFinder finder(dyn);
5   AttractorRange ar = allStates | finder;
6   Counter<Attractor> c(ar);
```

Like in the previous version, if we iterate on `Counter` we get all attractors with their respective basin sizes.

Derrida plot. A useful mathematical tool to analyse BN dynamics is the Derrida plot [59], which is a graphical way to visualise the sensitivity of a network to perturbation, namely, bit-flips in its state. Intuitively, the Derrida plot depicts the qualitative response of a network when h of its nodes are flipped, $\forall h \in 0, 1, \ldots N$. More formally, the Derrida plot associates to each perturbation strength $h \in 0, 1, \ldots N$, reported in the x-axis, a quantity $d \in [0, N]$, reported in the y-axis. This quantity d is the average across all network states s of $d_H(F(s), F(s\prime))$, where F is the map associated to a BN, d_H is the Hamming distance and $s\prime$ is a perturbed version of s, specifically, $d_H(s, s\prime) = h$. Some versions of the Derrida plot report h and d normalised by the number of nodes N so that $0 \leq h, d \leq 1$. One remarkable property of Derrida plots is that the slope at the origin of the Derrida curve is equal to average network sensitivity [129].

The brute-force computation of a Derrida plot is, of course, prohibitive for all but the smallest networks. Nevertheless, we can resort to approximation and, for each data point in the plot, take the average on a sample of network states. Let us show ho we can employ the Boolean Network Toolkit to calculate the data points in a Derrida plot, supposing that we set the number of sample for each data point to M.

```
1   BooleanNetwork bn = // read BN from file
2   BooleanDynamics* dyn = synchronous_dynamics(bn); // get an
        updater
3   std::vector<double> derrida(N);
4   for (int x = 0; x < N; ++x) {
5     std::vector<int> distances(M);
6     for (int i = 0; i < M; ++i) {
7       State s = random_state(N);
8       State sPrime = random_flips(s, x); // x random flips
9       State s1 = (*dyn)(s);
10      State sPrime1 = (*dyn)(sPrime);
11      distances[i] = hamming_distance(s1, sPrime1);
12    }
13    derrida[x] = average(distances);
14  }
```

Aside from `average`, all other functions are implemented in the toolkit; specifically, `random_state` generates a state vector whose element have value 1 with probability 0.5 while `random_flips` returns a state with x randomly chosen bits flipped (`hamming_distance` has intuitive meaning). This shows how the toolkit can be easily extended with new experiments. As a matter of fact, all use cases presented in this section are already implemented in the toolkit by ready-to-use functions.

8.4 Conclusions and discussion

In this section we motivated the need of a robust, flexible and efficient BN simulator software and introduced the Boolean Network Toolkit, a free open-source BN simulator. We gave an overview of its design principles and commented its modular and extensible architecture. Finally, we presented some use cases, taken directly from the source code.

The Boolean Network Toolkit is actively used and maintained and a couple of important extensions are also planned. Having encapsulated network dynamics in a class allows us to freely define different update schemes without modifying other code. It is sufficient to implement the interface in Listing 8.1 for all update schemes desired. Although not currently available, implementations of different update strategies, such as asynchronous and random, are on their way. Another extension regard the integration of other network types, in particular, we plan on adding two alternative BN models, namely, Boolean Threshold Networks [107, 180] and Glass Networks [124].

Chapter 9

Designing Boolean Networks by Metaheursitics

The main objective of this this part of the thesis is to outline a methodology capable of automatically designing Boolean networks given a set of desiderata. For example, this set of desiderata can be a collection of conditions on the attractor landscape, such as the number of such attractors, the distribution of their period or their basin structure. These requirements are appropriate in the context of the ensemble approach, where we want statistical real-world features of a system (eg., number of cell types) to closely mirror model features (eg., number of attractors) of a family of Boolean networks. Other kinds of desiderata may involve the fulfilment of some static goal. For instance, we will see how in Section 9.3 we will ask for BNs that can reach in a certain time a specific target state. In one last kind of scenario, we might instead utilise a BN as a black-box learning system that, when asked a question, suitably codified as a Boolean vector, it is able to give a meaningful answer. This chapter presents one problem for each scenario described so far.

9.1 Introduction

As written in Chapter 7, BNs have been mainly considered as GRN models, enabling researchers to achieve prominent results in the field of complex systems biology [4, 199, 202, 208]. Nevertheless, in spite of their similarities with neural networks, their potential as adaptive task-focused systems has not yet been fully investigated and exploited. In this section, we first summarise the works in the literature that concern training or automatically design BNs; then, we illustrate our method and describe four applications. Although they may seem quite abstract, the case studies we are going to present are fundamental as a way to test the applicability of our methodology to the ensemble approach. We know that in the ensemble approach we are interested not in the specific details of some real biological system, but rather in their statistical features (see Section 7.2). The abstract properties we require from resulting BNs do not depend on a specific model or real-world experimental data, and allow us to test the ensemble approach in a generic setting. Moreover we recognise that cellular systems exhibit dynamical characteristics, such as evolvability and

insensitivity to small external perturbations, that are desirable when designing artificial systems. A methodology that automatises the construction of models of biological systems could, thus, be useful also in the design of robust and adaptive software systems.

The first one is the problem of designing networks with a prescribed attractor length (Section 9.2); this is more an introductory application that aims to illustrate in a concrete case study our methodology and outlines some difficulties we had to address. In Section 9.3 we tackle the problem of controlling a BN's trajectory to match a target state; this problem can be a stepping stone in the application of BNs as controllers and exemplifies the application of analytical tools typical of the metaheuristic domain, such as runtime distribution and landscape autocorrelation analyses.[1] Sections 9.4 and 9.5 are respectively devoted to the analysis of the landscape of attractors in RBNs with respect to similarity metrics, and to the automatic design of BNs with maximally dissimilar attractors. Finally, Section 9.6 takes on the Density Classification Problem, a task previously applied in the cellular automata domain, with the aim of showing the learning capabilities of BNs.

9.1.1 Related work

The first work concerning BNs as learning systems has been presented by Patarnello and Carnevali [173] who trained, by means of Simulated Annealing [132], a feed-forward Boolean network to perform binary additions. A study on the automatic design of BNs appeared the same year and was proposed by Kauffman [126]. The goal of the work was to generate networks whose attractors matched a prescribed target state. The algorithm proposed is a sort of genetic algorithm with only a mutation operator (no crossover) that could either randomly rewire a connection or flip a bit in a function's truth table, and extreme selection pressure: once a fitter individual was found it would replace the whole population. In a sense it is similar to a stochastic ascent local search. Lemke et al. extend this scenario [140] in that they require a network to match a target cycle. In this work a full-fledged genetic algorithm (with crossover) is employed. Another evolutionary approach is adopted by Easmaeili and Jacob in [76], where they require a population of RBNs to maximise a fitness function defined by a combination of several feature like network sensitivity, number of attractors and basin entropy. In their algorithm, a network can undergo changes in both functions and topology. Notably, mutation operator was allowed to add or delete a node. Their study is limited to networks of small size ($N \leq 10$).

Several works addressing evolution of robust BNs have been proposed by Drossel and others. In these works, robustness is intended as the capacity of a network to return to the same attractor after a random flip occurs in one of its nodes. Various search strategies have been employed and will be outlined in the following. In [221] the authors applied a stochastic ascent (called "adaptive walk" in the paper) to networks with canalising functions; the move operator could rewire a connection or replace a function with a canalising one. The next three contributions revisited and extended this last paper, with the same goals of finding robust networks. Mihaljev [158], instead of a local search, proposes

[1] Data collected in Section 9.3 are reproduced from Mattia Manfroni's master thesis [150], with his permission. Data are included in this dissertation because they are subject of further discussion which arise from the search space analyses performed.

a genetic algorithm whose mutation operator is the move procedure described above. Fretter [81] studies the dynamical properties of evolved networks with *any* functions, not only canalising ones. Szejika [222] extends her previous work and this time investigates the behaviour of evolved networks with Boolean threshold functions. In a work by Espinosa-Soto and Wagner [77], populations of random Boolean threshold networks (a special case of RBNs) are evolved, by means of a genetic algorithm, to investigate the relationship between modularity and evolvability of GRNs. The actual algorithm utilised is a simple genetic algorithm with constant size population, no crossover and a mutation operator capable of modifying edge weights in the topology graph. Finally, we mention a work in which probabilistic BNs are trained so as to learn the equality function [64].

To summarise, the techniques proposed in the literature to train a BN belong to either of two families: local searches (stochastic ascent and variants, Simulated Annealing, etc.) and Genetic Algorithms (with variants in the genetic operators). The methods used are quite simple and might not be effective in tackling hard learning tasks. Indeed, the goals addressed in the literature are mainly concerned with investigating phenomena in evolutionary biology, rather than tasks in machine intelligence. We believe that the recent advances in engineering stochastic local search can be fruitfully applied also in the task of training and designing BNs. In this paper, we show that advanced local search strategies, as well as algorithm engineering and analysis, enable us to design BNs for accomplishing difficult learning tasks.

9.1.2 Methods

The paradigm we adopt in this work is that of supervised learning. We suppose it is possible to define an objective function that evaluates the performance of a network with respect to a given task to be accomplished. Besides the objective function, the learning process requires that some parameters of the system can be subjected to variations according to a learning algorithm. Since there are no dedicated algorithms for general BNs, we formulate the learning process into an optimisation problem. A prominent example of this approach is that of evolutionary robotics, in which evolutionary computation techniques are used for designing robots controlled by neural networks [165]. In this perspective, the learning process of a BN can be modeled as a combinatorial optimisation problem by properly defining the set of decision variables, constraints and the objective function. We should also remark that, always in the context of evolutionary robotics, the objective function is rather a guide to the underlying solving procedure than an actual measure of the quality of a solution (which, in fact, might not be precisely measurable).

In our approach, which is illustrated in Figure 9.1, the metaheuristic algorithm manipulates the decision variables which encode the node transition functions of a BN. A complete assignment to those variables defines an instance of a BN. This network is then simulated and evaluated according to the specific target requirements. A specific software component is devoted to evaluate the BN and returns an objective function value to the metaheuristic algorithm, that, in turn, proceeds with the search. Therefore, the evaluation provides the feedback used in the learning process.

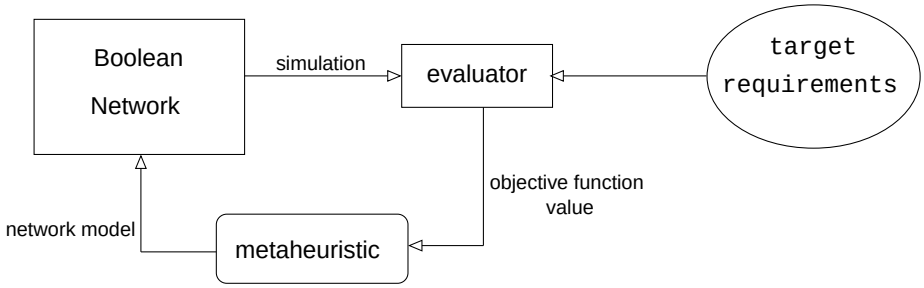


Figure 9.1: Scheme of the process for training a BN. The BN is simulated and its behaviour is compared to the desired one defined by specific requirements. The *evaluator* component provides a feedback to the metaheuristic, which manipulates the BN parameters.

In this thesis we investigate the topic of automatic design by employing population-based metaheuristics and trajectory methods. Specifically, Genetic Algorithms are the metaheuristic of choice for the design of BNs with desired attractor periods (Section 9.2). GAs will also be used as term of comparison in the last task studied, the Density Classification Problem in Section 9.6.

In all other problems under consideration, a specific metaheuristic has been used, namely Iterated Local Search (ILS), which extends the basic perturbative search. ILS is a well-known algorithmic framework, illustrated in Algorithm 9, successfully applied to many hard combinatorial optimisation problems [53, 144]. ILS makes it possible to combine the efficiency of local search with the capability of escaping from the basin of attraction of local optima. ILS applies an embedded stochastic local search method (Line 6) to an initial solution until it finds a local optimum; then it perturbs the solution (Line 5) and it restarts local search.

Algorithm 9 Iterated Local Search high-level framework.

1: INPUT: A LOCAL SEARCH
2: $s \leftarrow$ generateInitialSolution()
3: $s_{best} \leftarrow$ localSearch(s)
4: **while** termination conditions not met **do**
5: $\quad s' \leftarrow$ perturbation(s_{best})
6: $\quad s'_{ls} \leftarrow$ localSearch(s')
7: $\quad s_{best} \leftarrow$ acceptanceCriterion(s_{best}, s'_{ls})
8: **end while**
9: **return** s_{best}

In this work we implemented the following choices to instantiate the ILS framework.

Acceptance criterion: accept a new solution if it is better than the current best one.

Perturbation: for each node function a random flip in the truth table is performed. This choice makes ILS not too close to random restart, while

keeping the perturbation computationally fast and easy to implement. As a drawback, local search moves can undo such perturbation, albeit unlikely.

The last component to be defined is the embedded local search procedure. We opted for Stochastic Descent (SD), a very basic search strategy, which, despite its simplicity, proved to be very effective. SD is a problem-independent local search algorithm whose pseudo-code is shown in Algorithm 10.

Algorithm 10 Stochastic Descent.

```
1: INPUT: A SOLUTION s, AN OBJECTIVE FUNCTION F, A NEIGHBOUR DEFINITION 𝒩
2: s_best ← s
3: ν ← 𝒩(s_best)
4: repeat
5:    randomly pick a neighbour s_0 ∈ N without replacement
6:    if F(s_0) < F(s_best) then
7:       s_best ← s_0
8:       ν ← 𝒩(s_best)
9:    else
10:      ν ← ν \ s_0
11:   end if
12: until timeout or ν = ∅
13: return s_best
```

In order to apply this algorithm to our task, we need to instantiate its problem-dependent components: the solution representation, that is, a state in the search space, the objective function and a suitable neighbour definition.

Search state: a search state is a BN.

Initial solution: a Random BN with defined N, K and bias p.

Neighbour definition: this component defines the modification, or *move*, performed on the current solution. For the experiments described throughout the chapter, the move used behaves as follows: first we randomly choose a node function, then we flip a bit in its truth table. The pair $\langle \text{node}, \text{truth table position} \rangle$ is uniformly sampled without replacement.

It should be noted that this definition of move does not dramatically change the network functions, therefore, a network in some dynamical regime has, with high probability, neighbours in the same regime.

The objective function depends on the specific task to be accomplished, therefore it will be described separately for each case study presented. In the problems that will be discussed, the goal of the search is to minimise the objective function, which can thus be considered as an error or a distance function.

9.2 Designing Boolean networks with prescribed attractor periods

In this section the first case study concerning automatic design of Boolean networks is presented. Our goal is to investigate the possibility of evolving BNs by GAs so as to obtain a network able to reach an attractor of a desired period with a trajectory starting from a given initial state s_0. This represents just one of numerous examples of requirements we may want a BN to satisfy. Nevertheless, since attractor length depends on the main properties of BNs, this goal enables us to address some of the relevant issues in BN design. The questions that we want to address are the following:

a) Is it possible to guide evolution in such a way to succeed in the goal? What is the probability of reaching the target? (I.e., how robust is the automatic design procedure?)

b) Are there differences across network parameters? Are there networks that are *easier* to evolve?

c) Which are the most difficult or the easiest targets to be reached?

d) What is the influence of GA parameters?

In the remainder of this section we detail the experimental settings and report and discuss the experimental results.

This section present the work of a study published on the proceedings of the 11^{th} conference of the Italian Association for Artificial Intelligence (AI*IA) [181].

9.2.1 Experimental settings

Experiments are run with networks of $N = 100$ nodes and $K = 3$. This value for connectivity is the minimum such that it is possible, by choosing an appropriate bias, to have an initial populations of RBNs from the three regimes: ordered, critical and chaotic. The initial state is randomly sampled in the space and the target attractor lengths are 1, 10, 50, 100, 500, 800. Networks composing the initial population are constructed according the RBN model: inputs are randomly assigned, without self-loops in the topology; Boolean functions are defined by assigning truth values biased by homogeneity values p equal to 0.85 (ordered), 0.788675 (critical) and 0.5 (chaotic), in three different experiment series, respectively. However, Boolean functions homogeneity of single individuals can change during evolution because the initial distribution of 1s and 0s can be changed by the genetic operators.

According to the guideline in Section 4.3.2, we specify the characteristics of this GA implementation.

Chromosome structure. The individuals of the GA are encoded as a tuple of N truth-tables, i.e., binary vectors of size 2^K, each defining the Boolean function of a node. Thus, only node transition functions of a network are evolved and the topology is kept constant during the evolutionary process.

Recombination operators. The recombination operators are the well-known one-point crossover and the single-variable flip mutation. Both operators are applied to each element of the chromosome with probability m_r/N (mutation) and c_r/N (crossover). Mutation and crossover rates, respectively m_r and c_r, are reported in Table 9.1.

Table 9.1: Summary of experimental parameter values. All possible combinations of the values reported have been tested.

N	K	p	attractor length	population size	number of generations	mutation / crossover rate	number of runs
100	3	0.5 0.788675 0.85	1 10 50 100 500 800	80	200	0.5 / 0.9 0.5 / 0.0 0.1 / 0.9	100

Fitness function. The fitness function of a network n is defined as $F(n) = (1+|l-l_t|)^{-1}$, where l is the length of the attractor the individual network reached and l_t is the target length. For efficiency reasons, the temporal evolution of each network is simulated for at most 1000 steps: if an attractor is not reached in this limit, a fitness value of 0 is returned.

Population update. Each generation, a new offspring population of the same size as the original population is generated. We select the best individuals according the steady-state strategy.

The remaining parameters of the GA have been chosen as reported in Table 9.1, in which a summary of experimental parameter values is provided. All possible combinations of the values reported have been tested.

BNs have been simulated with the Boolean Network Toolkit and the GA has been implemented with GAUL [1].[2] All experiments have been performed on a 2.4 GHz Intel Core 2 Quad with 4MB of cache and 2GB of RAM, running with Linux Ubuntu 8.10.

Performance comparison

We first discuss the results concerning the performance of each class of networks, addressing questions *(a)*, *(b)* and *(c)*. The first notable observation is that *for all* target attractor lengths and *for all* initial network classes, the GA could find at least one network with maximal fitness in the 100 independent runs. This result means that all three classes of networks can be evolved to successfully reach the target. To assess the robustness of the process, we compare the fraction of successful runs at each generation of the algorithm, i.e., we estimate the *success probability at generation t*, defined as the probability that a network with maximal fitness is found at generation $t' \leq t$. The corresponding plots are depicted in Figures 9.2 and 9.3. Results for attractor lengths of 1 and 10 are omitted, because the fraction of runs achieving maximal fitness reaches the 100% right in the initial population or after few generations.

We first note that the performance achieved with initially ordered networks is considerably lower than that of critical and chaotic ones. This can be ascribed to the fact that ordered networks are not very likely to have long attractors. Anyway, the search process performed by the GA is still able to find a network with the desired attractor length. The case of critical and chaotic networks

[2] We could not employ `EasyGenetic` because this study was made before a functional version of our framework was available.

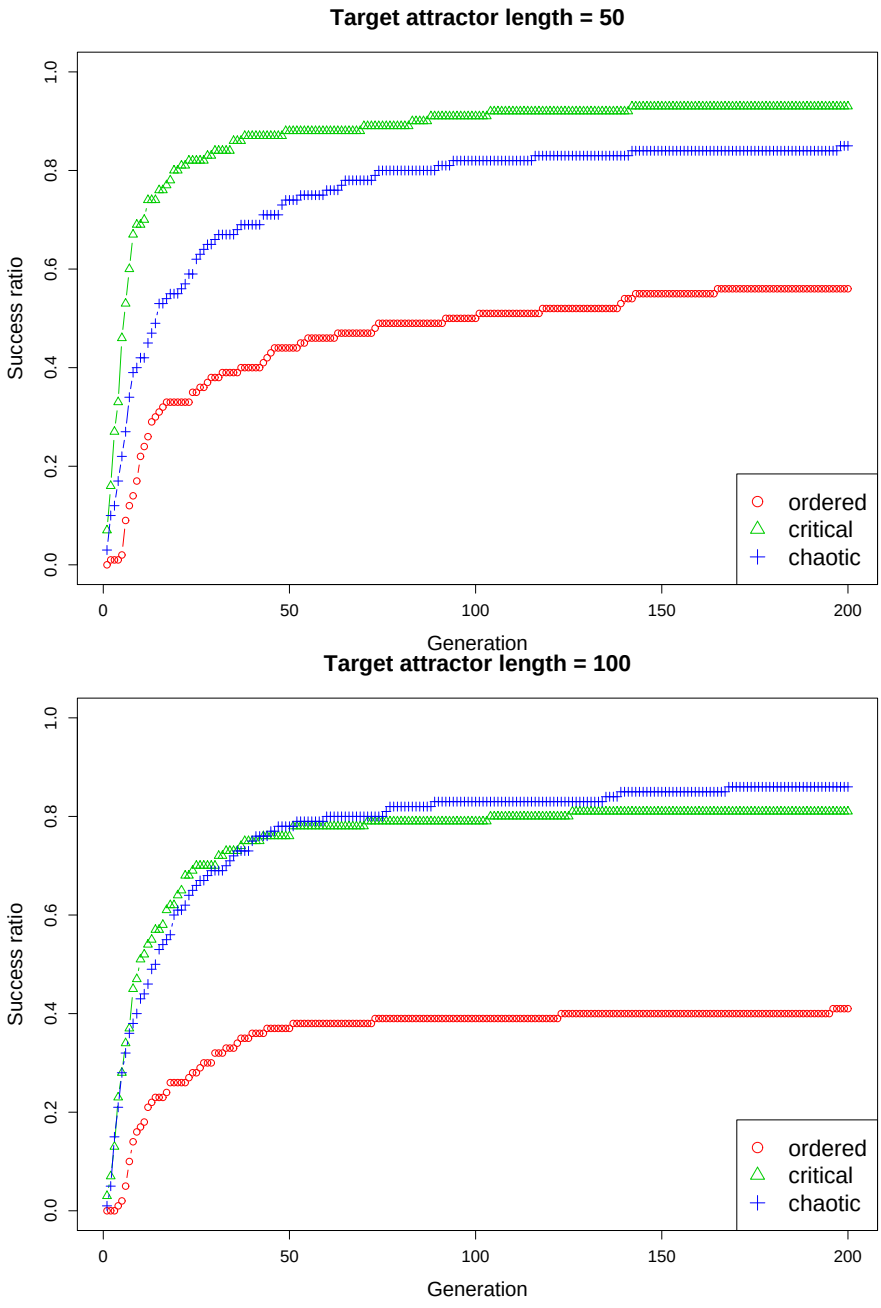


Figure 9.2: Success ratio vs. generations. The comparison is made among the three initial network classes. Target attractor lengths equal to 50 and 100.

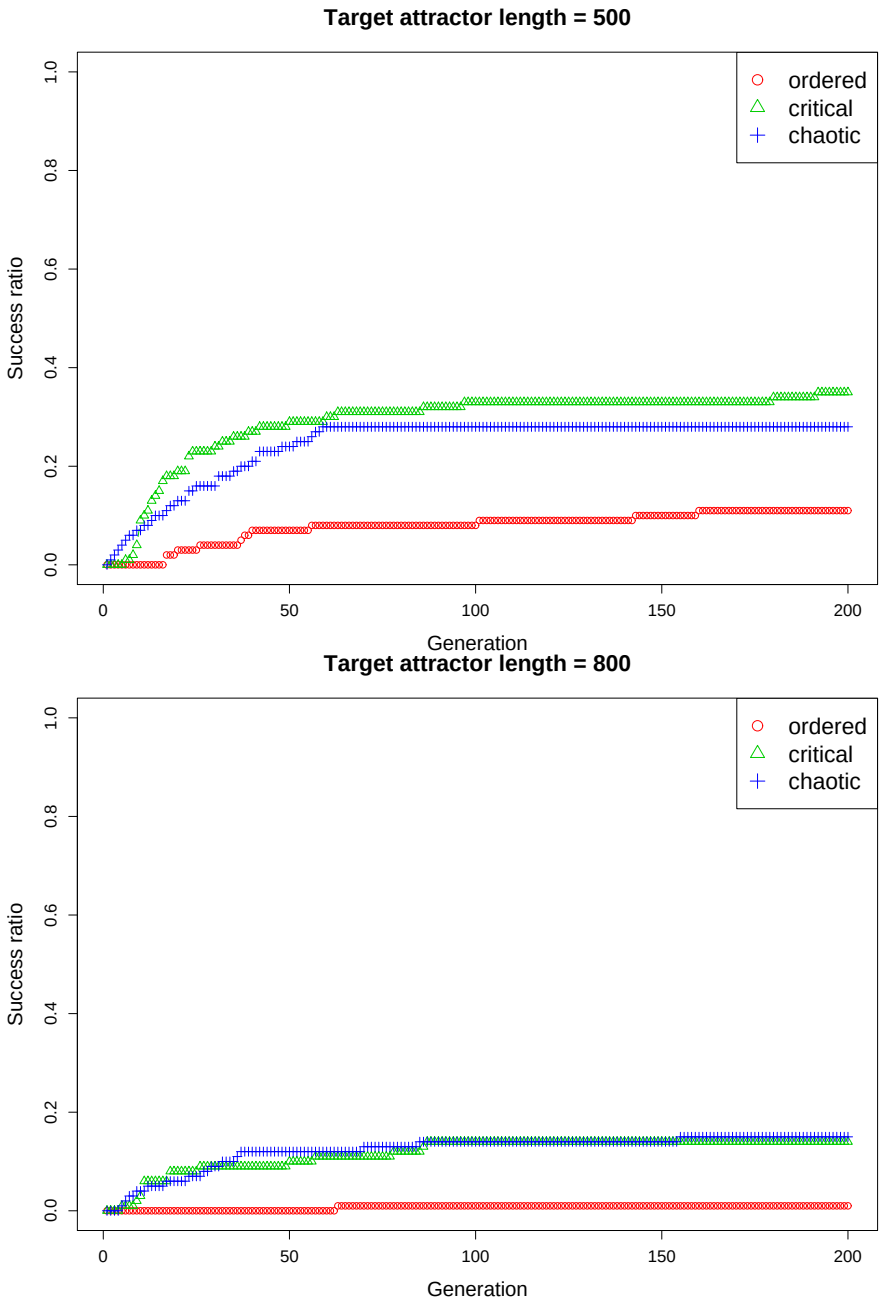


Figure 9.3: Success ratio vs. generations. The comparison is made among the three initial network classes. Target attractor lengths equal to 500 and 800.

has some subtleties which deserve to be outlined. First of all, we observe that the success ratio decreases as the attractor length increases. Moreover, in most cases critical networks dominate or are almost equivalent to chaotic ones, while for target attractor length equal to 100, initial chaotic networks seems to provide a better start to the GA. Both the phenomena can be explained by the combination of two factors. First: the cutoff imposed on simulation steps limits from above the networks attractor length, hence making it difficult to evolve networks with an attractor of length comparable with the maximal number of simulation steps because, if an attractor is not found, the corresponding fitness value is zero. Second: critical networks have usually many attractors, but of small length compared to attractor periods of chaotic networks, that can be exponential in the number of nodes. In a survey experimental analysis, we observed that for networks with 100 nodes and a maximal number of simulation steps of 1000, the median attractor length for critical networks is 6, while for chaotic ones is 130. Therefore, for a target length of 100, the fitness of individuals composing the initial population is likely to be higher in the case of chaotic networks than in critical ones. However, it is worth to be noted that critical networks can be anyway evolved to reach long attractors, despite their handicap in the initial population's fitness. This could be a further evidence of their tendency of maximising adaptiveness.

The study of the search space, that would provide insight into problem hardness, is introduced and partially addressed in Sections 9.3, but, overall, is still subject of ongoing work.

Influence of GA parameters

The influence of mutation and crossover on search performance can shed light on the evolution characteristics of the different initial population classes and can answer question *(d)*. Figures 9.4, 9.5 show a typical case[3] of algorithm performance in the three examined cases of mutation and crossover rates. From the plots we observe that the synergy of both mutation and crossover are crucial for the evolution of initially ordered and critical networks. Conversely, for chaotic networks, mutation is much more important than crossover.

In these experiments, we have shown that it is possible to evolve ensembles of networks with a desired attractor length regardless of the dynamical class of the initial networks. It would be possible, of course, to experiment with other targets, such as specific patterns in the attractors and combinations thereof, aiming at the design of networks with a desired landscape of attractors, each with a specific characteristic. Although in this dissertation we do not directly address this problem in general terms, a similar task is tackled in Section 9.6, since, as we will see, we ask for networks with structured basins of attraction and attractors with specific characteristics.

Another possible extension could involve the relaxation of the constraint of keeping constant the initial state, thus moving to stochastic search problems. We also notice that in this case the evaluation would be more complicated because, since would be dealing with a stochastic object, we would need to calculate a meaningful statistics (confront the beginning of Section 4.3.1) to obtain a fitness value. Such statistics usually involve a large number of samples

[3]Target attractor length equal to 100.

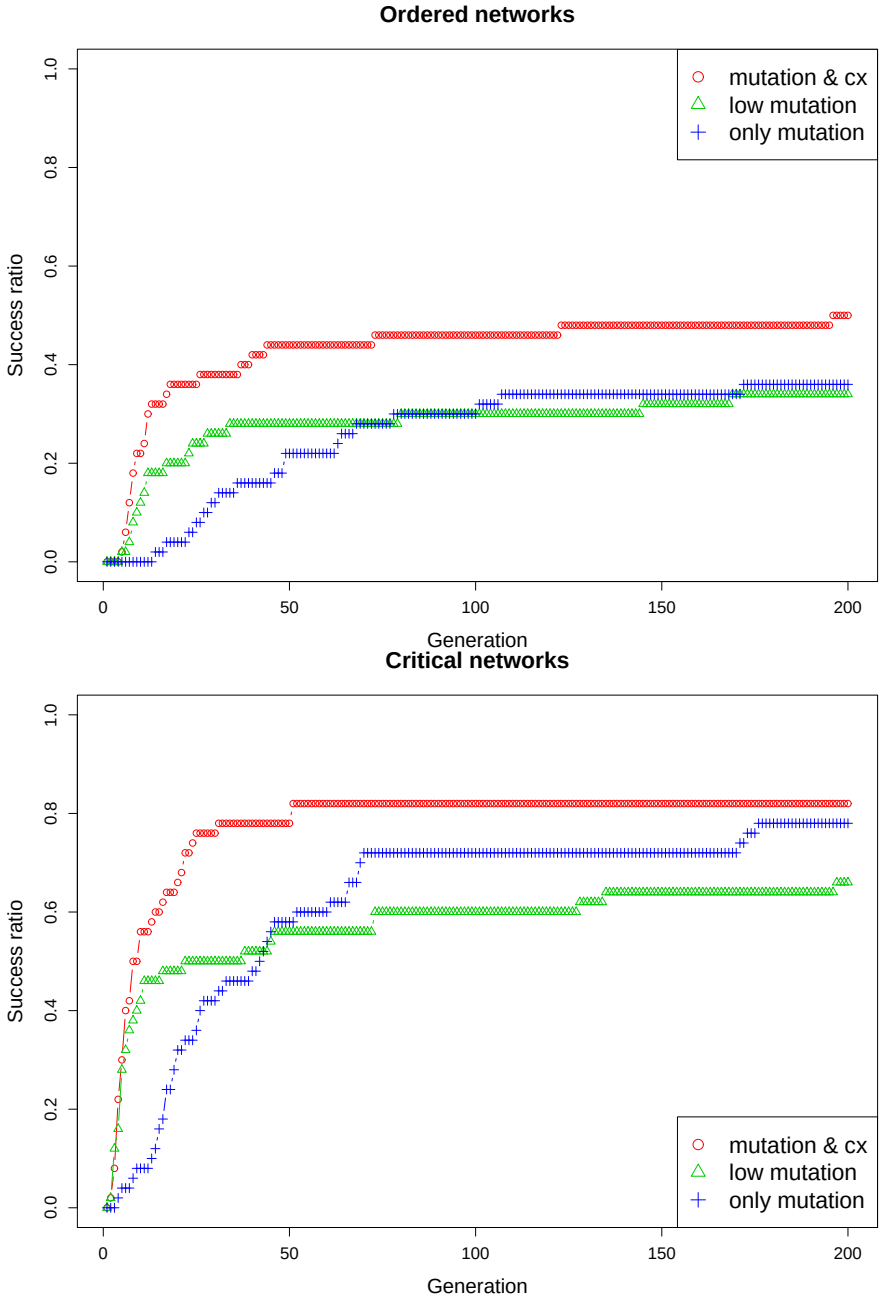


Figure 9.4: Comparison of the impact of mutation and crossover on search performance. The case of ordered and critical initial network classes are reported.

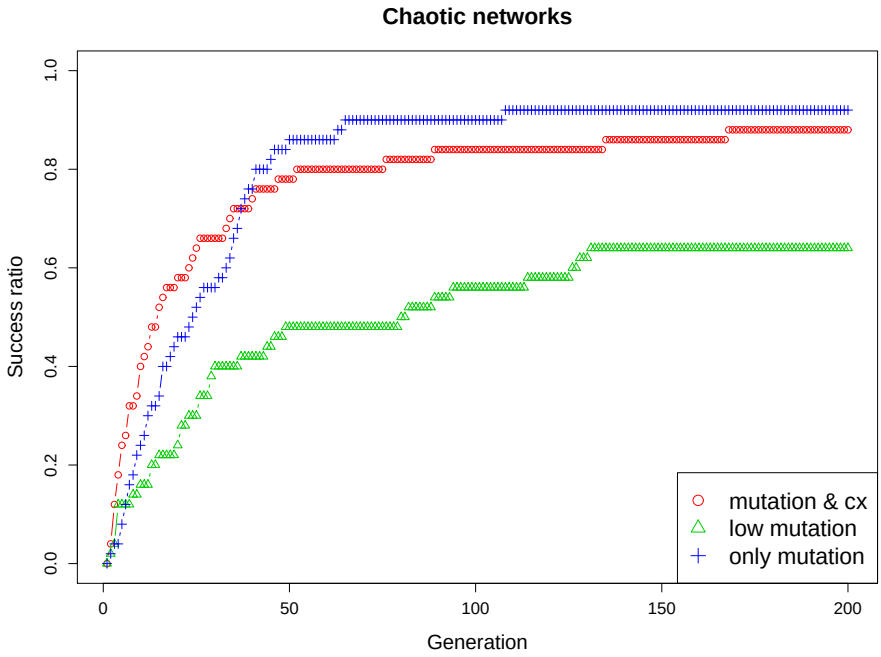


Figure 9.5: Comparison of the impact of mutation and crossover on search performance. The case of initial chaotic network class is reported.

from the configuration space and therefore a large number of simulations. This poses a heavy computational burden on the experiments. Moreover, it is difficult to determine whether such statistics is accurate since in all but the most trivial cases, the configuration space is remarkably undersampled. This is the typical scenario in the experiments that follow.

As a final remark, in this preliminary case study, we faced many interesting questions regarding the search process worth investigating, such as: *(1)* the relationship between search space and network characteristics, primarily topology and initial dynamical class. For instance, it would also be interesting to witness a correlation, or a lack thereof, between the dynamical class of the networks in the initial population and the specific target. We could find, for example, that seeding the algorithm with critical networks, as opposed to chaotic ones, could favor the search on a particular target, or, on the contrary, our GA is efficient enough for a wide range of targets. *(2)* The characterisation of the networks obtained at the end of the search process, specifically, which measures should we adopt to describe such networks since they were definitely not generated according to the RBN model but, on the contrary, have been subject of an evolutionary process. A possible answer is given at the end of Section 9.5.

9.3 Target state-controlled Boolean networks

In this section, we describe the experiments in which we train a BN in such a way that some requirements on its trajectory are fulfilled. The problem of

designing a dynamical system such that its trajectory in the state space satisfies specific constraints is a typical control problem. The general problem of network controllability has been studied in [143] with one important difference. In the paper a system is controllable if there exist a sequence of inputs values that drives the system from *any* initial state to *any* desired target state in finite time, which is a much stronger requirement than the one we consider in this section. Controllability of Boolean networks has also been extensively studied by Cheng et al. [50, 51].

In the case of BNs, which exhibit in general complex dynamics, this task we tackle is not trivial for an automatic procedure, because an assignment of Boolean functions has to be found such that the resulting BN dynamics fulfils the requirements. This problem has been chosen with the aim of assessing the effectiveness of our approach. In fact, for a BN with N nodes and K inputs per node, the whole search space has a cardinality of $2^{2^K N}$.

In this section, we are concerned with tasks in which a target state has to be reached, subject to additional constraints. Given are an initial state s_0, a target state $\hat{s}$ and a number of network simulation steps T. The goal is to design a BN such that the trajectory in the state space with origin in s_0 reaches the target state $\hat{s}$ in one of the following conditions:

1. the target state $\hat{s}$ is reached in a number of steps less than or equal to T;
2. the target state $\hat{s}$ is reached for the first time at step t, such that $t \in [z, T]$, $0 < z < T$, where z is a parameter of the problem;
3. as point 2, but with the additional requirement that the target state is a fixed point.

Experiments and data presented in this section are reproduced from Mattia Manfroni's master thesis [150] because they are matter of further discussions which arise from the search space analyses performed.

9.3.1 Experimental setting

The BNs used for this task have $N = 100$ nodes and $K = 3$ distinct inputs per node (no self-connections). The connections of the networks are randomly generated as per RBN model. The initial Boolean functions are also randomly generated with bias $p \in \{0.5, 0.788675, 0.85\}$ correspond to chaotic, critical and ordered regimes, respectively. For each value of p, 30 RBNs have been generated.The initial state s_0 of each BN is generated according to a uniform distribution over the whole state space. The target state $\hat{s}$ is likewise randomly generated.

The metaheuristic search used for designing the BN is ILS, which has been described in Section 9.1.2 with a slight variation in the move definition. Since the goal, in all the three cases, consists in matching a given target state, the neighbourhood of the current solution can be sampled with a heuristic bias, trying to focus the search on promising neighbours of the current solution. To this aim, the node function to be changed is chosen among the ones corresponding to nodes whose values do not match the target state. The rationale behind this heuristic is that of "repair algorithms", in which local search moves only affect the parts of the current solution that contribute to increasing the objective function (to be minimised).

For each experiment, 100000 iterations of the optimisation algorithm have been executed. Each iteration corresponds to a simulation of the respective BN trajectory lasting T steps, with $T = 1000$ (i.e., 1000 BN state updates).

Task 1: reaching a target state

The goal of this case study is to train a BN in such a way that its trajectory reaches a given a target state $\hat{s}$ at least once within the temporal left-open interval $]0, T]$. The evaluation of a BN is done on the basis of the evolution time step in which the BN presents the largest number of Boolean variables matching the target state. Let $u(t)$ be the function returning the number of Boolean variables matching the target state at each simulation step t, with t belonging to the left-open interval $]0, T]$; the objective function, to be minimised, can be described as follows:

$$\min_{t \in]0,T]} \left(1 - \frac{u(t)}{N}\right).$$

To assess the robustness of the process, we compute the fraction of successful runs at each iteration of the algorithm, i.e., we estimate the *success probability at iteration* t, defined as the probability that a network with minimal objective function is found at generation $t' \leq t$. This kind of statistic is also known as *run length distribution* [110]. The results obtained are shown in Figure 9.6. We first note that all the BNs with initial bias $p = 0.85$ reach the goal within 80000 iterations of the optimisation algorithm. Also BNs initially in critical regime present good performances, whereas only 10% of chaotic BNs reaches the goal. A reason for explaining this phenomenon could be related with the properties of the initial networks. BNs generated with $p = 0.85$ are very likely to be ordered, therefore small changes in their Boolean functions correspond to small variations in the objective function. Conversely, most of the BNs generated with $p = 0.5$ behave chaotically; slightly differing chaotic networks have a very different behaviour, similarly to what we have described in Section 7.1.1 concerning perturbations in the initial state of RBNs. Therefore, the initial search space is smooth in the case of ordered networks, whilst it is rather rugged for chaotic ones. The difference in the performance of the learning algorithm when starting from ordered vs. critical networks[4] can be ascribed to the fact that critical BNs are known to exhibit a mixture of characteristics typical of ordered and chaotic BNs (see also Section 9.4 for a further example). This conjecture on the ruggedness of the search landscape can be tested by estimating the *autocorrelation* of the landscape. Smooth landscapes are characterised by high autocorrelation, while rugged ones have low autocorrelation [112]. The autocorrelation of a series $G = (g_1, \ldots, g_m)$ of objective function values is computed as

$$r = \frac{\sum_{k=1}^{m-1} (g_k - \overline{g}) \cdot (g_{k+1} - \overline{g})}{\sum_{k=1}^{m} (g_k - \overline{g})^2},$$

where $\overline{g}$ is the average value of the series. This definition refers to the autocorrelation of length one, i.e., that corresponding to series generated by sampling 1-Hamming neighbouring states in the search space. For each network's dynamic class we computed the empirical autocorrelation of 1000 time series obtained by

[4] generated with $p = 0.788675$

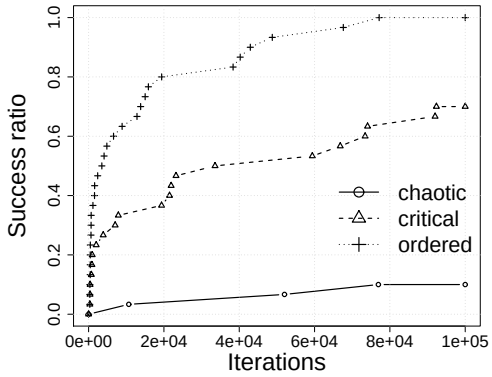


Figure 9.6: Run length distribution related to task 1: reaching a target state.

collecting the objective function values along a random walk of 100 steps starting from 30 randomly generated initial candidate networks for each value of p. The boxplots in Figure 9.7 summarise the statistics of the values of autocorrelation in the three dynamical regimes. We note that the landscape of ordered and critical networks is highly correlated ($r \approx 0.9$), whilst the one of chaotic networks is not ($r \approx 0.4$). This result supports our hypothesis for explaining the different performance across network's dynamical regime. It is important to stress that our finding shows that the very reason for a different behaviour is to be ascribed to the combination of two factors: the dynamical regime and the objective function used in the learning task. As we will see in Section 9.6, the objective function may be a crucial factor in inducing different landscape correlation properties.

Task 2: reaching a target state within a given time window

The goal of this second case study is to design a BN whose trajectory reaches a given a target state $\hat{s}$ at least once within the temporal interval $[z, T]$, but not before z, with $z \in]0, T]$. In this case, the objective function should be defined with care. Indeed, it is important that the function not only reaches its minimum if the constraint on the trajectory is reached, but it should also guide the search toward the satisfaction of such constraint. Therefore, we assign a certain reward also to those BNs whose trajectory reaches a state either almost congruent to the target one or a certain number of simulation steps τ before z. To implement this reward rule, we define a family of functions $f(t;\gamma)$ on the interval $[0, T]$ as follows:

$$f(t;\gamma) = \begin{cases} 0 & t < z - \tau \\ 1 - \left|\frac{t-z}{\tau}\right|^{\gamma} & z - \tau \leq t \leq z \\ 1 & t > z \end{cases} .$$

The function $f(t;\gamma)$ is plotted in Figure 9.8: note that BN states before z can be rewarded in diverse ways, depending on the value of parameters γ and τ. Let $u(t)$ be the function returning the number of nodes matching the target state

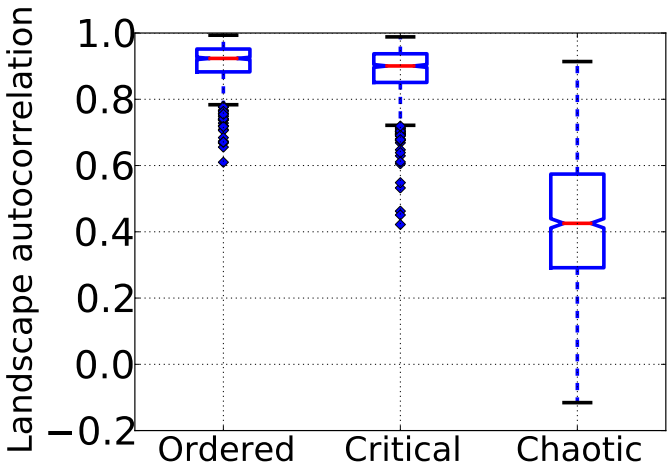


Figure 9.7: Distribution of autocorrelation r of the landscapes corresponding to networks in different dynamical regimes.

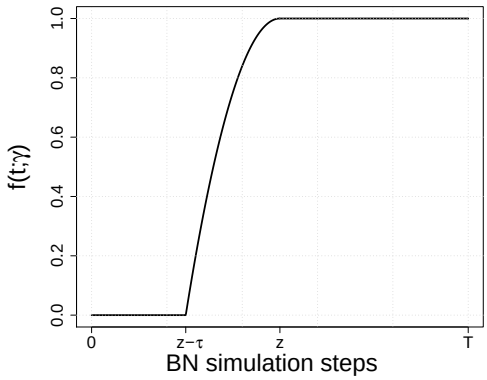


Figure 9.8: Function $f(t;\gamma)$ used to assign a reward to partially successful BNs. In the figure, the function with $\gamma = 2$ is plotted.

at each simulation step t, with $t \in]0, T]$, the objective function can be described as follows:

$$\begin{cases} 1 & \text{if } \frac{u(t)}{N} = 1,\ t \in]0, z-\tau] \\ 1 - f(t;\gamma)\frac{u(t)}{N} & \text{otherwise} \end{cases} . \tag{9.1}$$

Note that this objective function does not reward at all those BNs whose trajectory reaches the target state before $z - \tau$.

In Figure 9.9, we show the results attained with $z = 500$ and adopting the following parameter setting: $\tau = 10$ and $\gamma = 2$. Analogous results have been obtained with $z = 50$ and $z = 100$, the case of $z = 500$ being the hardest

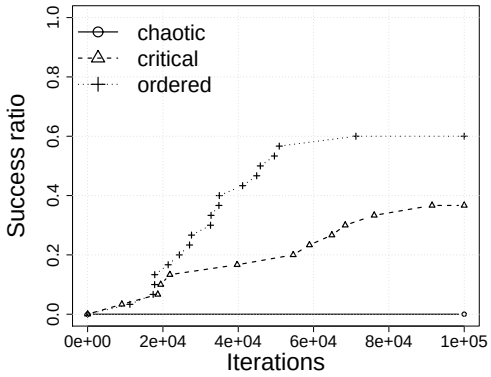


Figure 9.9: Run length distribution related to task 2: reaching a target state within a given time window.

for the learning process. In addition, different values of τ and γ have been tested ($\tau \in \{10, 20, 50, 100\}$ and $\gamma \in \{0.5, 1, 2\}$) and no statistically significant difference was observed.[5]

The performances attained in this task are qualitatively the same as for the previous case study, even if the overall performance is lower with respect to the previous case. This is not surprising, as this task is rather more difficult than the previous one. Also for this task we computed the empirical autocorrelation of the search landscape and we obtained the same qualitative results as in the previous case.

Task 3: reaching a fixed point target state within a given time window

The goal of this third case study is the same as the previous one, with a further constraint: when a BN trajectory reaches the target state, then such state must be kept. In other words, the target state must be a BN fixed point. As in the previous case study, we assign a certain reward also to those BNs whose trajectory reaches a state either almost congruent to the target one or a certain number of simulation steps τ before z. At each network evaluation, let $z' \in [z - \tau, T]$ be the simulation step corresponding to the state with the largest number of Boolean variables congruent to the target state. To verify that BN state in z' is a fixed point of the BN, it is enough to check if the state at $z' + 1$ is equal to the one in z'. If this occurs, then we can assert that the BN trajectory has reached a fixed point. This statement is valid because we are considering BNs with deterministic dynamics and synchronous state updates.

Analysing the requirements, two different main features can be noticed: first of all, the network has to reach the target state, but not before $z - \tau$. To evaluate this aspect, we can use the same objective function as in the previous case study, which is defined by Equation 9.1. The second issue consists in making the target state a fixed point for the BN. A way to merge these two aspects is to define an

[5] We applied both the χ^2 test and the Fisher's test [57] to the success percentages and the null hypothesis (i.e., equal distributions) could not be rejected.

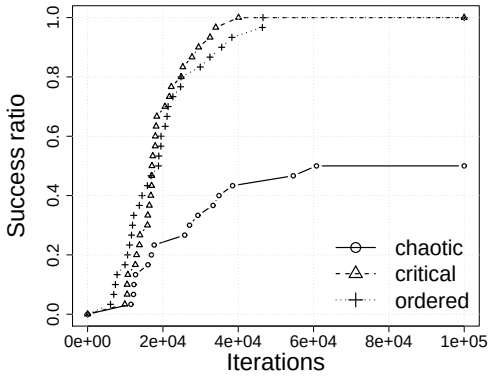


Figure 9.10: Run length distribution related to task 3: reaching a fixed point target state within a given time window.

objective function based on a weighted mean, as follows:

$$\begin{cases} 1 & \text{if } \frac{u(t)}{N} = 1,\ t \in]0, z-\tau] \\ \alpha x(z') + (1-\alpha) y(z') & \text{otherwise} \end{cases} .$$

where $x(z')$ is defined by Equation 9.1 and $y(z')$ is a function that compares the BN states in z' and $z'+1$, returning the ratio between number of not congruent Boolean variables and the total number of BN nodes. Thus, when $y(z') = 0$, the BN state in z' represents a fixed point for the BN.

Different values for α account for different relative importance between reaching the target state and keeping it. We tried several values of this parameter—$\alpha \in \{0.25, 0.5, 0.75\}$—to estimate its impact on the optimisation process. We noticed that small values of α (i.e., $\alpha \leq 0.5$) lead to a slightly better performance than the one attained with $\alpha = 0.75$. In Figure 9.10, we show the results obtained with $\alpha = 0.5$ and other parameters set to the same values as the results showed for the previous case study. We can note that the overall performance is better than in the previous case. We conjecture that the behaviour of the local search is positively affected by the introduction of the objective function component accounting for the fixed point constraint. In fact, once a BN is tuned such that a fixed point is reached, it is not hard to further change the Boolean functions so as to match the target state. This conjecture finds an independent support in a recent work in evolutionary robotics [34].

The next two sections are part of a study of attractor features of the classic RBN model. In Section 9.4 we present an experimental study in which we show relevant statistical features of the similarity among attractors in RBNs. This research will point out a limitation of RBNs when used as models of real GRNs, a limitation that, specifically, is tied to the synchronous update scheme and to the high similarity between attractors. In Section 9.5 we will instead apply our methodology to partially overcome this limitation; our aim will be, in fact, to engineer networks with an attractor landscape as varied as possible.

9.4 Attractor distances in Boolean networks

A prominent feature that can be considered in the ensemble approach is the distribution in distances between gene expression levels in different types of cells. In Section 7.1.1 we anticipated that attractors in RBNs can be made correspond to cellular types [127], a conjecture further refined in terms of *threshold ergodic sets* [198, 228]. This extension provides support to the effectiveness of RBNs as genetic regulatory network models, as it makes it possible also to model cell differentiation dynamics. In order to test this conjecture, two issues have to be addressed: first, the properties of RBN attractors have to be studied; second, these properties have to be compared with the ones of cellular types. In this thesis, we aim at providing a contribution to the first issue by studying the statistics of distances between attractors in random Boolean networks.

This section is based on a paper by Roli et al. [182].

9.4.1 Attractor similarity statistics

In real cells, each type is characterised by a specific pattern of gene expression levels which can be represented as real number vectors of size N, where N is the number of genes. On the model side, we can make the hypothesis that each attractor of a BN represents a cell type. Statistics and, possibly, other kinds of information on the distances between attractors can be computed and then compared against equivalent statistics on gene expression levels in real cell types, so as to test to what extent the class of RBNs capture relevant properties of ensembles of real cells.[6]

In this section, we introduce the distance measures we defined over the attractors, along with the statistics and properties we analysed.

Attractor distance measures

We defined and studied three different distances among BN attractors, namely *Minimum Hamming* (MHD), *Euclidean* and *pseudo-Hamming*. The first one is defined upon the states composing the attractors, while the other two are defined upon the average values of BN nodes in each attractor.

Minimum Hamming distance (Definition 5) measures the minimum number of node values that should be changed in order to let the network's trajectory jump from an attractor directly to another one.

Definition 5 (Minimum Hamming (MHD))**.** Let A_i and A_j two attractors of a Boolean network, their Minimum Hamming distance is:

$$d_{MHD}(A_i, A_j) = \min\{d_H(s, s') \mid s \in A_i, s' \in A_j)\}$$

where $d_H(s, s')$ is the Hamming distance between states s and s'.

It is important to observe that this distance does not depend on the network dynamics in the state space, as it simply considers the Hamming distance between states independently of the state space trajectory. Measures which depend on the actual state space topology can be also defined.

[6] The hypothesis of the correspondence between attractors and cell types is therefore operational, rather than ontological.

The following distances are defined over real vectors $V(A_i) = \langle v_1, \ldots, v_n \rangle$, each one computed for a given attractor A_i. Elements v_j $(j = 1, \ldots, n)$ are computed by averaging the values assumed by variable x_j along the attractor, i.e., by computing the fraction of times a Boolean variable assumes value 1 along the attractor, or, in other words, computing the time average of its values along the attractor (see also Section 9.6 for a similar application). In formulas: given attractor $A = (s^{(1)}, \ldots, s^{(\tau)})$ of period τ, with $s^{(h)} = \langle x_1^{(h)}, \ldots, x_n^{(h)} \rangle, h = 1, \ldots, \tau$, each element v_j of vector $V(\mathcal{A})$ is computed as: $v_j = \frac{1}{\tau} \sum_{h=1}^{\tau} x_j^{(h)}$. It has to be noted that this mapping between attractors and vectors of real numbers makes it possible to establish a simple yet direct semantics of a BN attractor as a gene expression level array [201].

A straightforward way of measuring the distance between two real valued vectors is to compute their Euclidean distance. This distance induces naturally a distance over attractors:

$$d_{Eucl}(A_i, A_j) = d_{Eucl}(V_i, V_j) = \sqrt{\sum_{l=1}^{n} (v_{il} - v_{jl})^2}$$

The Euclidean distance might smooth the differences between expression vectors, thus making it hard to distinguish between attractors of different length. In fact, attractor cycles of very different length might be mapped onto real valued vectors whose Euclidean distance is very small. For this reason, we introduced a distance that is computed by summing up the number of homologous vector entries which are different, and we call it pseudo-Hamming distance.

Definition 6 (pseudo-Hamming)**.** Let A_i and A_j two attractors of a Boolean network, their pseudo-Hamming distance is:

$$d_{\psi H}(A_i, A_j) = d_{\psi H}(V_i, V_j) = \sum_{l=1}^{n} 1 - \delta(v_{il} - v_{jl}), \text{where } \delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

Attractors clustering

Given the attractors of a BN network, a *distance matrix* can be constructed according to the distances previously defined. Besides computing the main statistical parameters of such data, distance matrices have been also used in two kinds of analysis: (*i*) distribution in (weighted) clustering coefficient; and (*ii*) attractor dendrograms.[7]

Clustering Coefficient. The clustering coefficient C_i of a vertex i in a graph provides an estimation of the how much its neighbours tend to form a complete graph. For a non-weighted graph, the clustering coefficient C_i is equal to its maximum value 1 if neighbours of i form a complete graph, while it is 0 if neighbours of i are disconnected. The average of vertex clustering coefficient provides an estimation of how much a graph is characterised by clusters of vertices. Formally, a network clustering coefficient is:

$$C = \frac{1}{N} \sum_{i=1}^{N} C_i, \qquad C_i = \frac{n_i}{g_i}$$

[7]Preliminary results have been published in [187].

where n_i is the number of edges between neighbours of vertex i and g_i the maximum possible number of edges between them. It is also possible to extend the clustering coefficient definition to weighted graphs [246]; in this case, the greater the edge weight, the stronger is the intensity of the connection between the two vertices. Values used for computing this measure are taken from a network adjacency matrix $A = (a_{ij})$, where an element a_{ij} corresponds to the weight of the edge which has its tail in i and its head in j; $a_{ij} = 0$ if $i = j$ or edge (i, j) is not present. In formulas:

$$n_i = \frac{1}{2} \sum_{u \neq i} \sum_{\{v \mid v \neq i, v \neq u\}} a_{iu} a_{uv} a_{vi} \quad , \quad g_i = \frac{1}{2}((\sum_{u \neq i} a_{iu})^2 - \sum_{u \neq i} a_{iu}^2)$$

In our analysis, the weight of an edge that connects two vertices (attractors) is the (normalised) reciprocal of the distance between the attractors.

Dendrograms. A network attractor distance matrix can be also used to graphically represent clustered distribution of attractors. For each network, a dendrogram has been generated, which represents in a single data structure all the possible clusters of the elements in a set. Attractor dendrogram analysis yields a graphical representation of the tendency of the attractors to gather into clusters. The results we present are based on dendrograms constructed using 'single-link' algorithm [122].

9.4.2 Experimental analysis

In this section, we present the results of the experimental analysis performed. As usual, for all simulations we used the Boolean Network Toolkit. We analysed the main statistical parameters of the distances between attractors in RBNs with 70 nodes,[8] $k = 3$ and bias values such that the networks are in ordered ($p = 0.85$), chaotic ($p = 0.5$) and critical ($p = 0.788675$) phases. For each parameter configuration, 50 independent RBN realisations have been generated. Each network dynamics has been simulated for at most 10^6 steps, starting from 10^5 initial states picked uniformly at random in order to sample attractor cycles.

Distance measures statistics

A first analysis concerns the main statistical parameters of the distance matrices. Table 9.2 shows the minimum, maximum, mean, median, 1^{st} and 3^{rd} quartile values of such quantities. We can observe that the maximal distances, independently of the actual definition used, are in chaotic networks. Moreover, also the mean and median values of attractor distance in chaotic networks are considerably higher than those of critical and ordered networks. The differences between the last two classes are smaller than those with respect to chaotic ones, even though critical networks show a larger spread in values and higher average values.[9] It is remarkable to observe that the qualitative pattern is the same, independently of the distance measure.

[8] Networks' size was constrained by the very large computational time required for simulating chaotic networks of larger size.

[9] An exception to this observation is the median of the Euclidean distance, but differences are very small and not significant.

Table 9.2: Distance statistics.

(a) Minimum Hamming distance

Bias	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.5	1	5	9	9.03	12	29
0.788675	1	1	4	4.62	7	24
0.85	1	1	2	3.86	5	18

(b) Euclidean distance between activation vectors

Bias	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.5	0.00	0.43	0.90	1.23	1.80	4.77
0.788675	0.00	0.36	1.23	1.18	1.74	4.69
0.85	0.00	0.67	1.16	1.36	1.88	4.24

(c) Pseudo-Hamming distance between activation vectors

Bias	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.5	0	66	68	63.44	70	70
0.788675	0	3	12	13.61	22	51
0.85	0	3	8	8.35	10	27

Attractors clustering

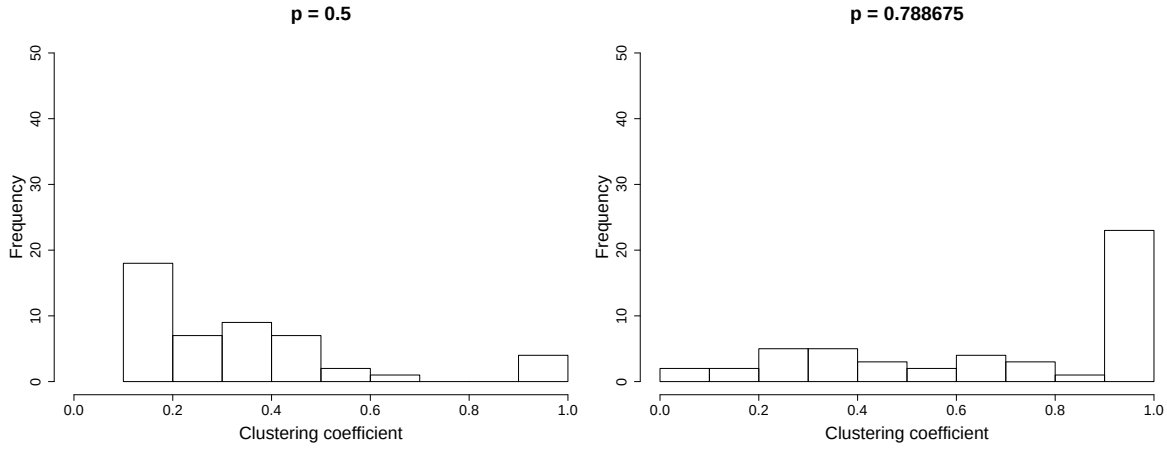
In Figures 9.11a, 9.11b and 9.11c, the histogram of the average clustering coefficient distribution is plotted for chaotic, critical and ordered RBNs, respectively. The distance measure considered is the MHD, but qualitatively analogous results have obtained also with the other distance measures. The pattern emerging from the histograms is not surprising: chaotic network attractors have a very low tendency of forming clusters, while in critical and ordered networks, attractors are clearly clustered. It is interesting to note that critical networks seem to exhibit a pattern that is a mixture of the chaotic and ordered ones, because the clustering coefficient distribution spans, with significant values, across the whole range.

A similar picture emerges from the dendrograms, which graphically capture the clusters emerging among attractors. In Figures 9.12a, 9.12b and 9.12c, typical cases of dendrograms for chaotic, critical and ordered BNs are respectively plotted.

9.5 Designing Boolean networks with maximally distant attractors

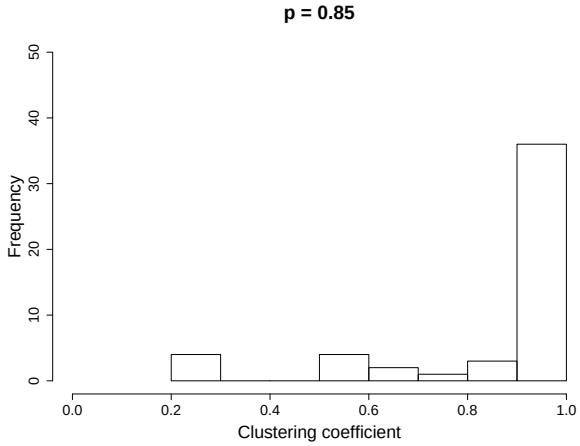
In this section, a natural continuation of the previous one, we are concerned with the problem of designing BNs exhibiting a set of attractors (i.e., long term behaviour patterns) as much diversified as possible.

This section is based on a work by Benedettini et al. [24].



(a) Chaotic RBNs

(b) Critical RBNs

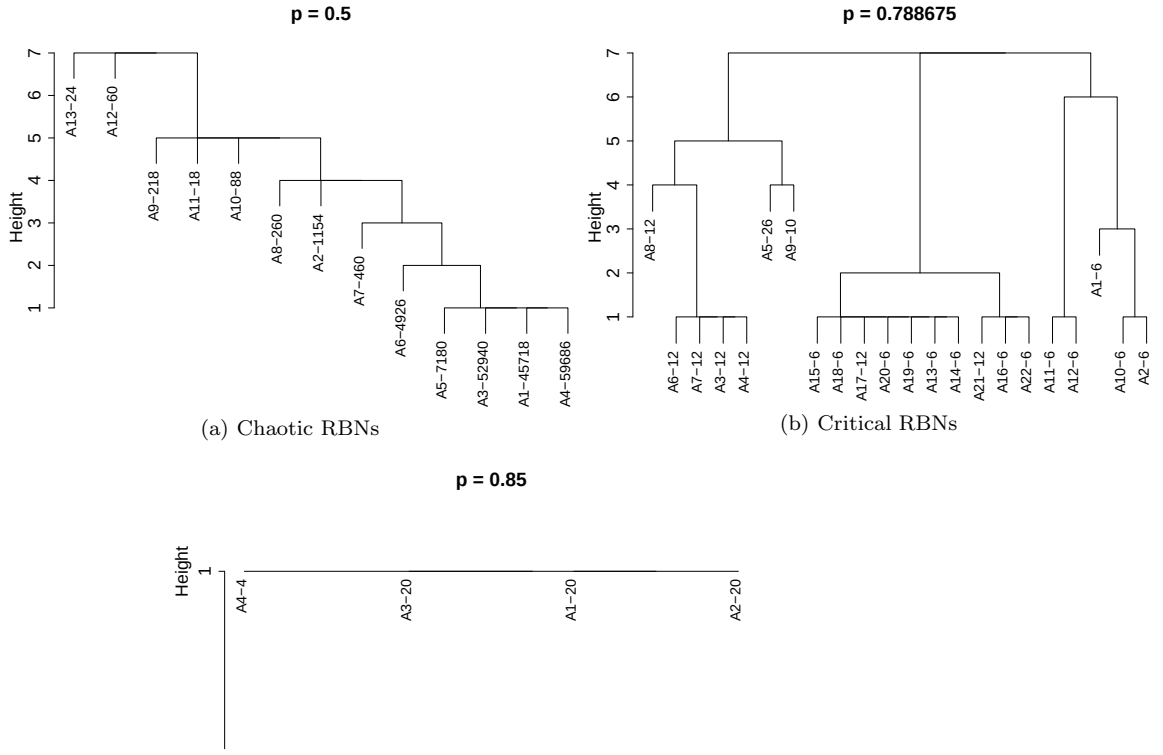


(c) Ordered RBNs

Figure 9.11: Average clustering coefficient distribution.

9.5.1 Objective and Motivations

As previously remarked, RBN models of genetic regulatory networks make extensive use of attractors. Nevertheless, as pointed out at the end of Section 9.4, the attractor set in synchronous and deterministic RBNs is very likely to contain attractors which differ for just a few values. These very same attractors are no longer distinguishable if a different update scheme is used, such as delayed update [97]. Therefore, synchronous and deterministic update could generate spurious attractors, which are meaningless from a biological perspective. However, most of the results in literature achieved so far assume this update scheme, which makes the dynamics of the network easy to analyse, both empirically and theoretically. In the following, by employing the methodology outlined in Section 9.1.2, we aim at designing synchronous and deterministic BNs in which attractors are as much different as possible so as to close the gap between RBNs



(a) Chaotic RBNs

(b) Critical RBNs

(c) Ordered RBNs

Figure 9.12: Typical samples of attractor dendrograms.

and more biologically plausible BNs. The possibility of designing such networks would also enable us to contrast their characteristics against those of RBNs, in the spirit of the ensemble approach.

Informally, our objective is to automatically design a BN whose attractors are as much different from each other as possible. In order to formalise this requirement, we need a definition of similarity. Toward this aim, we choose the Minimum Hamming distance between attractors as stated in Definition 5 in Section 9.4.1. We recall that MHD represents the minimum number of bit flips required to directly bring the dynamical state of a BN from attractor A to attractor B, or *vice versa*. We chose such measure because of its possible biological relevance; if we interpret attractors as cell type and bit flips as external noise [10], we can say that the probability by which an external, possibly harmful, perturbation switches a cell (BN) from type (attractor) A to type B is roughly inversely proportional to $d_{MHD}(A, B)$, or equivalently, their dissimilarity. Moreover, the conclusions drawn in Section 9.4 allow us to say that, to some extent, the results presented in this section are independent of the distance chosen. Experiments in Section 9.5.2 show that attractor distance distribution strongly depends on network dynamical regime and that different distance definitions

[10] An interpretation also consistent with the work by Serra et al. [198].

give rise to attractor distance distributions with similar statistical properties.

To be able to use our methodology, we have to define an objective function. For reasons explained later, we choose as our objective function the *expected MHD between attractors*, defined below.

Definition 7. Expected Minimum Hamming distance between attractors. Let $\{A_1, A_2, \ldots, A_h\}$ be the attractor set of a BN with (relative) basin weights $\{w_1, w_2, \ldots, w_h\}$. The expected MHD is:

$$E_d = \sum_{i=1}^{h}\sum_{j=1}^{h} w_i w_j d(A_i, A_j) = 2\sum_{i=1}^{h}\sum_{j=i+1}^{h} w_i w_j d(A_i, A_j)$$

This gives the expected MHD between two randomly sampled attractors. It is important to observe that the attractor landscape of an optimal network which maximises this objective function consists in two complimentary fixed points with basin weights equal to 0.5.

The expected MHD is difficult to compute except for the smallest networks because it requires the complete enumeration of the attractors of a network along with their basin weights, therefore we resort to approximation in order to improve the efficiency of the search. We use a Monte Carlo method to estimate the attractor set of a network and their basin weights. We start from m random initial states $S = \{s_1, \ldots, s_m\}$ for the network. For each state we evolve the network up to an attractor and we collect the attractors found $\{A_1, A_2, \ldots, A_h\}$. If we denote with $S_i \subseteq S$ the states from which the network reaches attractor A_i, the approximate basin weight $\hat{w}_i$ of an attractor A_i is $\frac{|S_i|}{m}$. By applying the formula in Definition 7, we have an approximation $\hat{E}_d$ for the expected MHD. In our experiments, we choose $m = 10^4$ random initial conditions.

Since we resort to random sampling, we incur in the problem first mentioned in Section 4.3.1 and restated at the end of Section 9.2: in order to measure the quality of a solution, we have to make an accurate estimation of its value across an adequate number of sample. To this aim, we decided to introduce weights in the distance definition in order to penalise attractors with a small basin size for both biological and practical motivations. Biologically, an attractor with a small basin corresponds to a rare event very unlikely to occur. Practically, this choice makes network evaluation much more robust because it reduces the variance of objective function estimation.

We conclude by saying that we also performed preliminary experiments both utilising average and median of attractor distance distribution as objective function, but with unsatisfying results.

9.5.2 Experimental Analysis

Experiments have been carried out starting from initial RBNs from different dynamical classes. In total, we generated 300 networks used as initial solutions: (*i*) 100 RBNs with $K = 3$ and $N = 20, 30, 70, 200$ nodes, bias $p = 0.211324, 0.788675$ (*critical ensemble*); (*ii*) 100 RBNs with $K = 3$ and $N = 20$ nodes, bias $p = 0.15, 0.85$ (*ordered ensemble*); (*iii*) 100 RBNs with $K = 3$ and $N = 20$ nodes, bias $p = 0.5$ (*chaotic ensemble*). It is known that chaotic networks have average attractor period that scales exponentially with system size [13]. This makes the simulation of chaotic networks much more

computationally expensive than for networks in ordered or critical regimes. If local search comes across a region of search space inhabited by chaotic networks, it will probably spend a great deal of available computation time, if not all. As already pointed out in Section 9.1.2, the single bit flip move implemented by Stochastic Ascent makes it so that that chaotic networks have, with high probability, chaotic neighbours; therefore, local search might not escape from that region and get stuck until time runs out. This issue could be partially addressed by imposing an upper bound on the number of simulation steps thereby drawing the exploration away from chaotic networks. We did not choose to implement such a constraint because we wanted to observe the behaviour of our algorithm in an un-biased environment. In the end, we decided to run experiments starting from networks in chaotic regime only for $N = 20$, in which simulation is still manageable. It is, of course, still possible that chaotic networks appear during search. Furthermore for $N > 20$ we start only from critical RBNs because several biological systems have been shown to operate in critical regime [10, 166].

In order to estimate the effectiveness of our method, we compare it against a simple Random Walk (RW) heuristic as a baseline. RW is the same procedure as SA with the only difference that moves are always accepted. This comparison is useful to roughly assess the benefits of a more sophisticated exploration of the huge network search space over an uninformed random walk.

We implemented our local search in C++ and employed the Boolean Network Toolkit as usual to implement function evaluation. Our program was compiled with gcc 4.4.0 with `-O3` optimisation. We executed the algorithm once for every network with a runtime limit of 7 hours. Experiments were performed on a cluster composed of quad core 2.33GHz Intel Xeon^{TM} processors with 12Mb of cache, 8Gb of RAM and running Cluster Rocks 5.3.[11]

Analysis of designed networks

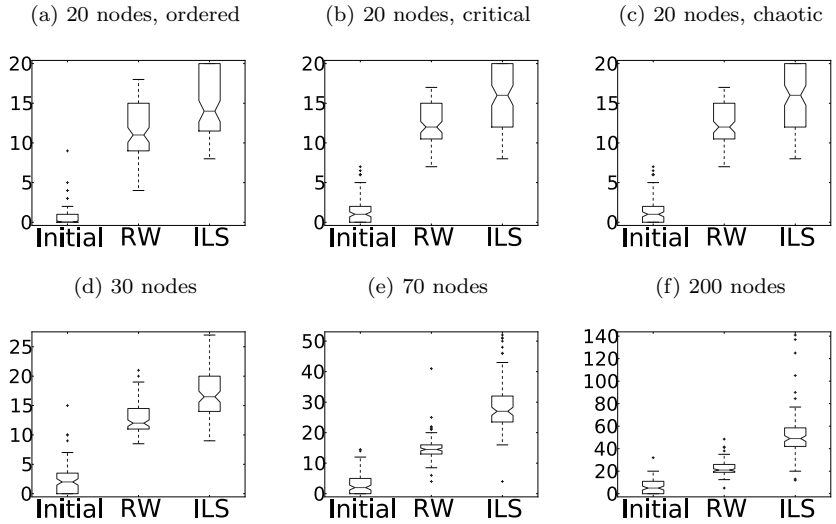
In the following, we illustrate the results achieved and we characterise the resulting networks from a dynamical point of view. The following analyses have been carried out for both initial and optimised networks. We simulated each network up to an attractor starting from 10^5 initial conditions picked uniformly at random and gathered the following data: distribution of the MHD between attractors, attractor periods, number of attractors and average network sensitivity. Such analysis is infeasible for chaotic networks with $N = 200$ due to computational difficulties. For such networks we limited ourselves to 10^4 samples and a subset of 20 RBNs. In the following, we compare the networks from the initial ensembles against the optimised networks returned by RW and ILS. For each network, we computed the median of the distribution of the MHD between attractors; the distributions of these values are summarised in the boxplots of Figure 9.13: in the top row there are networks with $N = 20$, in the bottom row networks with $N > 20$. Since the notches of the boxplots do not overlap, we can conclude that, with 95% confidence, the true medians of the distributions do differ [46]: ILS outperforms RW on all test cases.[12]

The dynamical analysis of optimised network is not a trivial task. As anticipated in Section 9.2, networks which undergo an evolutionary process exhibit

[11] `http://www.rocksclusters.org`

[12] We recall that these evaluations are based on random sampling.

Figure 9.13: Median of attractor distance distribution.



features from all dynamical classes and single measures, such as sensitivity, capable to classify a RBN, fail to fully capture the dynamical properties of evolved networks (see also [81]). We show that this is the case also for our experiments. The next tables summarises the distribution of attractor period, number of attractors and sensitivity for optimised and initial networks. Column headers report minimum, average, median, standard deviation and maximum. Table 9.3 compares networks with $N = 20$; initial RBNs belong to all dynamical regimes. Table 9.4 compares side-by-side chaotic and critical RBNs against optimised networks for $N > 20$. We can observe that the sensitivity of final networks is almost always greater than 1. A possible explanation is that, as the number of inputs increases, the fraction of functions with unitary sensitivity decreases [25]. For this reason, the search space explored by our local search has a lower density of network with sensitivity close to one. Although sensitivity is greater than one, that does not alone suffice to conclude that the resulting networks are in chaotic regime: the distribution of other measures is, in fact, different than that of chaotic RBNs, as shown in Tables 9.3 and 9.4. The most remarkable measure is attractor period: optimised networks have much shorter attractors than chaotic RBNs. This was indeed expected, because it is a direct consequence of the chosen objective function: an attractor with short period has less chances to have states close to those belonging to another attractor. Also, the number of attractors is slightly greater. *En passant*, we observe that the combination of these two properties (short cycles and high number of attractors) is typical of critical BNs.

Regarding the behaviour of the local search itself, in our tests we found that many ILS iteration were performed for small networks while only a few were performed for the largest networks. An ILS iteration is a succession of an intensification phase (embedded local search) and a diversification phase (perturbation). The explanation is that smaller networks not only require less simulation time to evaluate the objective function, but have also smaller neighbourhoods.

Table 9.3: Summary of network features for $N = 20$.

Class	Measure	Original RBNs					Optimised networks				
		min	μ	median	σ	max	min	μ	median	σ	max
Ordered	Period	1	2	1	2	12	1	1	1	0	5
	N. of Attr.	1	1	1	0	5	2	3	3	1	11
	Sensitivity	0.44	0.78	0.77	0.14	1.11	1.25	1.43	1.44	0.09	1.60
Critical	Period	1	3	3	3	35	1	1	1	0	4
	N. of Attr.	1	2	2	2	12	2	3	3	1	9
	Sensitivity	0.69	1.00	1.01	0.13	1.31	1.20	1.44	1.45	0.09	1.66
Chaotic	Period	1	8	4	11	66	1	1	1	0	3
	N. of Attr.	1	4	3	2	14	2	3	3	1	8
	Sensitivity	1.26	1.48	1.48	0.10	1.73	1.21	1.44	1.43	0.09	1.76

In a small neighbourhood it is less likely for SA to find an improvement, therefore the perturbation has higher chances to be applied more than once in the given time limits.

9.6 Density classification problem

In this section we take on the problem of training BNs to solve a classification problem. We know that cell behaviour can change in response to variation in the concentration of nutrients or other chemical substances: the cell adapts its own internal dynamics in reaction to different condition in the environment. From an abstract point of view, we can say that a cell is able to solve a classification problem, where the environmental conditions represent the example to classify and the resulting cell dynamical behaviour is the response. Since BNs are used to describe GRN behaviour, it is natural to ask whether BNs are able to learn how to perform an analogous function. In this section we focus on a particular classification problem (the Density Classification Problem, described below) and we ask whether it is possible to design a BN that, when subject to different initial conditions (corresponding to external stimuli in the case of real cells), it responds with a specific dynamical behaviour, i.e., attractor.

The Density Classification Problem (DCP), also known as Density Classification Task, first introduced by Packard, is a simple counting problem [171] born within the area of cellular automata (CA), as paradigmatic example of a problem hardly solvable for decentralised systems. Informally, it requires that a binary CA (or more generally a discrete dynamical system—DDS) recognise whether an initial binary string contains more 0s or more 1s. In its original formulation, the nodes (or cells) are arranged in a one dimensional torus and can interact only with the neighbouring ones. The problem is that of designing simple rules, governing the dynamics of each node, in such a way that the system is driven to a uniform state consisting of all 1s, if the initial configuration contains more 1s, or all 0s otherwise. In other words, the convergence of the DDS should decide whether the initial *density* of 1s is greater or lower than $\frac{1}{2}$.

Although the assignment might look trivial, it is a challenging problem and it is known for having no exact solution in the case of deterministic one-dimensional CA [138]. The origin of this difficulty is very intriguing and comes

Table 9.4: Summary of network features for $N > 20$.

N	Measure	Chaotic RBNs					Critical RBNs					Optimised networks				
		min	μ	median	σ	max	min	μ	median	σ	max	min	μ	median	σ	max
30	Period	1	18	6	36	335	1	3	2	3	20	1	1	1	1	9
	N. of Attr.	1	4	4	3	15	1	3	2	4	37	2	6	6	3	22
	Sensitivity	1.29	1.50	1.49	0.09	1.75	0.78	1.01	1.01	0.10	1.27	1.31	1.46	1.46	0.07	1.63
70	Period	1	1072	51	4426	79050	1	7	4	10	134	1	3	2	4	51
	N. of Attr.	1	6	6	2	14	1	4	2	4	28	2	9	9	5	31
	Sensitivity	1.38	1.50	1.50	0.05	1.61	0.87	1.01	1.01	0.06	1.14	1.06	1.33	1.34	0.11	1.57
200	Period	1	$6.8E5$	$1.4E5$	$1.2E6$	$7.6E6$	1	15	8	26	726	1	13	6	110	4445
	N. of Attr.	2	4	4	1	9	1	23	3	96	878	2	19	12	21	130
	Sensitivity	1.39	1.47	1.47	0.03	1.53	0.90	1.00	1.00	0.04	1.09	1.00	1.19	1.19	0.06	1.38

from the impossibility to centralise the information or to use counting techniques: the convergence to a global uniform state should be obtained by using only local decisions, i.e., by using just the information available in time within the close neighbours of a node. Given these difficulties, various modifications to the classical problem have been proposed, including stochastic CA, CA with memory, CA with different rules succeeding in time (see [78] and references cited therein). Interestingly, some authors directly investigated the dichotomy between the local nature of the CA and the global requirements of the related DCP by allowing the presence of long range connections within the links of the otherwise local neighbourhood [154, 197, 226, 234]. In particular, it can be shown that the simple *majority rule* applied on random topologies outperforms all human or artificially-evolved rules running on an ordered lattice [154, 197]; a performance gap that increases with the number of nodes [154]. The majority rule states that the value of a CA cell at time $t+1$ is 0 (resp. 1) if the majority of its neighbours has value 0 (resp. 1) at time t.

These last two cited studies demonstrate that RBNs can effectively deal with the DCP. Our aim in this section is that of demonstrating that learning RBNs are flexible objects, able to attain a performance comparable to a hard-to-match benchmark such as the majority rule. Therefore, we will not use extremely large neighbourhoods or network sizes, but rather we will focus our attention to the learning process itself, leaving scaling issues to further work.

In order to define the learning processes, we divide the nodes of a BN into three (possibly overlapping) groups: input nodes, output nodes and hidden nodes[13]. Of course, this separation is not sufficient to completely specify the overall learning scheme since there are many details regarding topology and node dynamics to be addressed. For instance, input nodes could maintain their initial values (this is the typical case in neural networks) or could evolve in time according to the typical BN dynamics; output nodes could have or not have feedbacks on the hidden/input nodes; moreover (see [6] and below), it is not clear what is the influence on the final attractors of the initial conditions of hidden and output nodes. A possibility, explored in previous studies [6, 186], consists in partitioning network nodes into input, hidden and output nodes. In this setting, the value of input nodes is externally imposed and does not change during network evolution, whereas hidden and output nodes are driven, as usual, by their transition functions. Nevertheless, in [6] it is also shown that different initial settings of hidden and output nodes typically lead to different attractors, making the analysis of the network's answer difficult. For the DCP we opt for an easier choice; we establish that: 1. all network nodes are input nodes; 2. all nodes are also output nodes (i.e., the state of each node contributes to the final answer); 3. there are no hidden nodes (it follows from the previous two conditions). This way there none of the nodes requires a special characterisation and the initial conditions are well defined. The correct answers can also be uniquely identified by two state vectors composed by all zeroes and all ones. Finally, and coherently with the Boolean nature of BNs, in order to correctly interpret oscillating asymptotic states it is enough to compute the time averages for each node, assigning "0" to the averages lower than 0.5 and "1" otherwise.

In this paper we use two groups of RBNs having respectively 11 and 21 nodes

[13] Although this categorisation is reminiscent of the distinction between input, hidden and output layers in neural networks, we have to remember that the topology of a Boolean network could be in principle any graph without any clear separation of node into "layers".

(odd numbers, as usual in the DCP, in order to avoid ambiguous situations where 0s and 1s are equally present), each with connectivity $K = 3$: this choice makes the formation and detection of local majorities possible. We create a training and a testing set for each $N \in \{11, 21\}$, assembled in order to uniformly sample the whole range of the density possibilities in initial condition vectors. Thus, in the training set, if N is the number of nodes, we have N vectors having one component set to 1 and the others set to 0, N vectors having two components set to 1 and the others set to 0, etc., up to N vectors having $N-1$ components set to 1 and the other one set to 0. To that, we add N vectors having all components set to 1 and N vectors having all components set to 0, for a total of $N(N+1)$ examples. This last addition emphasises the importance of giving a correct answer when the example coincide with one of the targets. The test set is similar, but lacks the first two and the last two series of the training set, in order to avoid useless reiterations of fitness evaluation. Therefore, the set is composed of $N(N-3)$ samples.

In order to employ a BN as a classifier, we need to specify some kind of procedure, a *classification function*, that maps an example to a class. In the following we detail such procedure. A network assigns examples to classes by following these steps. Let's call these classes σ_0 (majority of zeroes) and σ_1 (majority of ones). Given an example s, a network is evolved up to an attractor starting from an initial condition s; the temporal average of the attractor is computed and then the resulting vector is binarised with threshold 0.5, that is, values are rounded to the nearest integer:[14] a binary vector is so obtained (confront the distance definitions in Section 9.4.1). If this vector contains more zeroes than ones (respectively, more ones than zeroes) the network assigns s to σ_0 class (resp., σ_1 class).

9.6.1 Experimental setting

We carried out our experiments starting from networks generated according to the classical RBN model, i.e., with constant input connectivity $K = 3$ and given function bias p (see below), with $N \in \{11, 21\}$ nodes. BNs are labelled as *critical*, *ordered* or *chaotic* according to the function bias values used to generate them. Details on network parameters are given in the following:

Critical ensemble: for each number of nodes $N \in \{11, 21\}$ and for each function bias $p \in \{0.211324, 0.788675\}$ we generated 50 networks for a total of 200 RBNs.

Ordered ensemble: for each number of nodes $N \in \{11, 21\}$ and for each function bias $p \in \{0.15, 0.85\}$ we generated 50 networks for a total of 200 RBNs.

Chaotic ensemble: for each number of nodes $N \in \{11, 21\}$ we generated 100 networks with function bias $p = 0.5$ for a total of 200 RBNs.

These networks are the initial solutions in our local search algorithm and will be collectively referred to as *initial set*.

The local search used for training the BNs is ILS, described in Section 9.1.2. After preliminary experiments on a randomly selected subset of networks, we

[14] Values equal to 0.5 are rounded to 1.

determined the termination criterion for the local search, which is also the only parameter to configure. We decided to stop our ILS algorithm after 150000 networks have been evaluated. Within this limit, we observed that the local search reaches stagnation. At each improvement over the previous *incumbent*—i.e., the currently best found solution—we recorded the node functions of the new best solution.[15]

The objective function evaluates a BN classifier on the training examples. The objective function is to be minimised and has values in the $[0, 1]$ interval. The evaluation process is remarkably similar to the definition of classification function given at the end of paragraph 9.6, up to the computation of the binary vector: we initialise the network to be evaluated with an example s, we evolve it to an attractor A, we compute the temporal average of A and binarise it with threshold 0.5 obtaining a Boolean vector v. The contribution of $s \in \{0, 1\}^N$ to the objective function is the following: if s belongs to σ_0 (resp., σ_1) the objective function value is $\frac{w_h(v)}{N}$ (resp., $\frac{N - w_h(v)}{N}$), where $w_h(v)$ is the Hamming weight of vector v—i.e., the number of '1' entries—and N is the network size. The contributes of all training examples is added up and divided by the number of examples seen. Notice that this definition entails that if the objective function is 0 then the BN classifier correctly classifies all examples in the training set. The converse however is not true: a BN might correctly classify all training examples even if its objective function is greater than zero. This definition of objective function rewards answers v different than the admissible ones (vectors of all zeroes and ones); this was done in order to better guide the search process and avoid the presence of large plateaus in the search space due to a (excessively) strict objective function. Suppose that, for example, a network, presented with the example $s = 0111010 \in sigma_1$, answers $v = 110110$[16]; the answer is "close enough" to the correct one 1111111 so the network should be rewarded.

In order to measure the classifying capabilities of the optimised networks, we compared them to BNs with a very specific structure. We thus generated a new ensemble, labelled *benchmark set*, whose networks have random topology, input connectivity equal to 3 and whose node functions are all equal to the Boolean majority function on three inputs. The benchmark set contains 100 BNs with $N = 11$ nodes and 100 BNs with $N = 21$ nodes.

9.6.2 Results

In this section we outline the analysis performed on data gathered from the experiments. We perform two kinds of analysis. First, we assess the classification error of our optimised networks and we compare it with the BNs in the benchmark set (Section 9.6.2). Secondly, we analyse the network generated during the search process and show to what extent selected network features are affected by the optimisation algorithm (Section 9.6.2). Finally, we compare the performance of our ILS with a genetic algorithm (Section 9.6.2).

[15] We recall that the local search move does not change network topology.

[16] We assume that v is the binarised temporal average of some attractor; this is not relevant to the aims of this example.

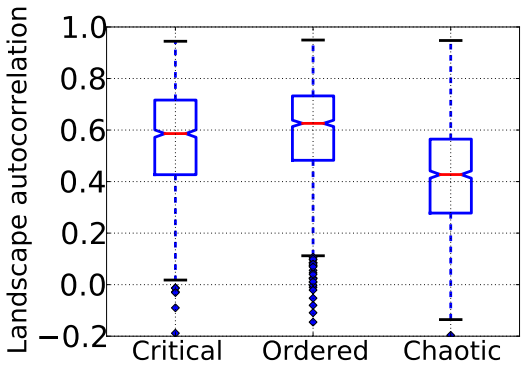


Figure 9.14: Distribution of landscape autocorrelation r for RBNs with $N = 21$ in different dynamical classes.

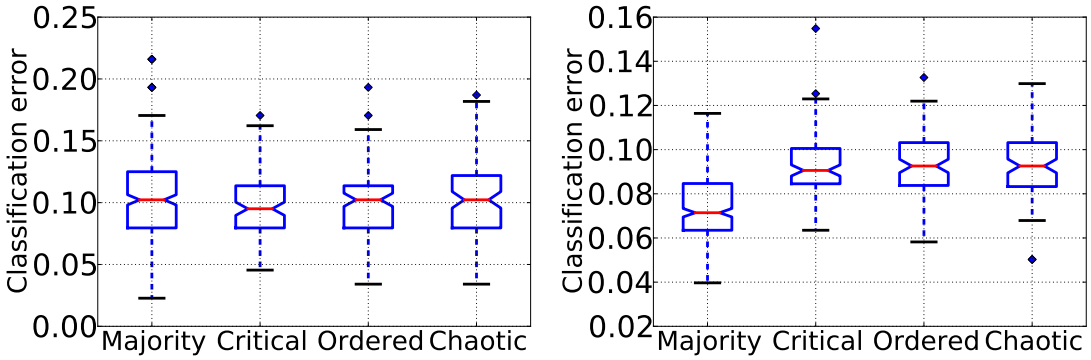
Performance analysis

The performance of the optimised networks is measured by the *classification error*, that is, the fraction of misclassified examples.

Figure 9.15 depicts the distributions of the classification error on the test set. These graphics compare the networks in the benchmark set (leftmost boxplot) with the classifiers generated by our metaheuristic starting from networks in the critical (second boxplot), ordered (third boxplot) and chaotic (last boxplot) ensembles. Performances of optimised networks do not significantly differ if the local search starts from either ensemble, although the distribution of the error for the chaotic ensemble has the lowest minimum (for $N = 11$ it ties the minimum on the ordered networks). We analysed the autocorrelation of the search landscapes, as done in Section 9.3. The boxplots showing the main statistics of the estimated autocorrelation of the landscapes in the three dynamical regimes are depicted in Figure 9.14. As in the previous test case, the autocorrelation of the landscape corresponding to ordered and critical BNs is higher than that of chaotic BNs. Nevertheless, the median autocorrelation coefficient r is rather low, ranging in $[0.4, 0.6]$ across all dynamical regimes. In addition, both the difference among the medians is quite low and the extreme values span across wide, overlapping, ranges. This result explains why, in this case study, we do not observe a difference across network's dynamical regimes as striking as in the case discussed in Section 9.3.

The remarkable performance gap between the benchmark and ILS observed when going from 11 to 21 nodes, can be explained by the fact that, as previously noted, the majority rule cannot be improved and its classification performance increases with the number of nodes [138, 154]. As a matter of fact, Figure 9.15 shows that performances attained by ILS with both network sizes are similar (the median for $N = 11$ is actually slightly lower).

It is also important to mention that even this optimisation scheme might be subject to *overtraining*. In this context, overtraining means that the network returned by our local search might not be the classifier that achieves the smallest classification error on the test set. Figure 9.16 shows two typical examples of overtraining. These graphics depict the classification error on both training and



(a) $N = 11$ (b) $N = 21$

Figure 9.15: Comparison of optimised networks against benchmark networks. Boxplots show distribution of classification error (fraction of misclassified examples) on the test set.

test sets attained by the *current best* network for every iteration of the local search. The training error curve is non increasing, as expected, while the test error curve has a global minimum before the local search reaches its locally optimal solution.

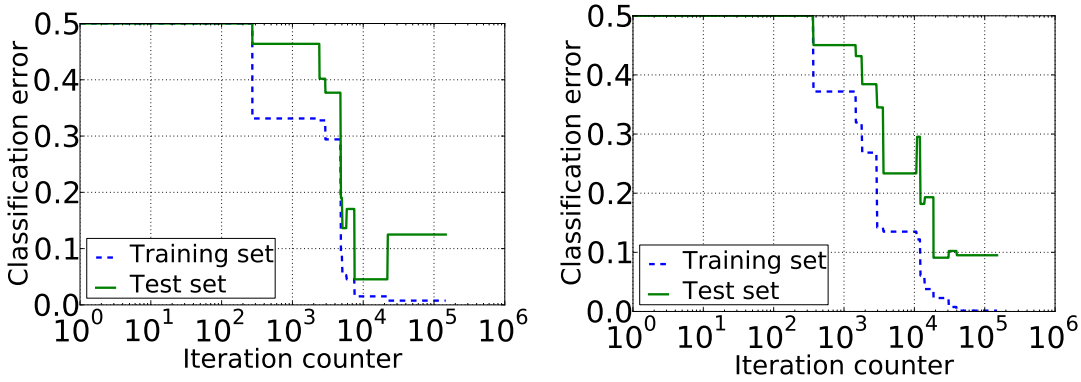
For this reason, for every algorithm execution we kept track of all the networks generated whenever a new local optimum was found; that way we could choose the BN which minimises the classification error on the *test set*.

Analysis of learning process

In the following, we analyse statistical properties of the networks generated by the search process. Specifically, we compare the initial networks in each ensemble with the best classifiers returned by our local search, i.e., the best BNs that achieve the smallest error on the test set (henceforth referred to as "*optimised networks*").

The comparison is performed on the following network measures: number of attractors, average attractor period, average network sensitivity and *pattern distance*. In order to calculate the former two measures, we simulate a network up to an attractor starting from all the possible initial conditions.

The *pattern distance* measures the average similarity of the node functions to the Boolean majority function and is computed as follows. Let us denote with $v_i \in \{0,1\}^8$ the truth table of the i-th node transition function; we compute the average over all nodes $\overline{v} = \frac{1}{N}\sum_{i=0}^{N} v_i$. We obtain a new binary vector $\pi \in \{0,1\}^N$ by rounding $\overline{v}$ to the nearest integer (0.5 is rounded to 1). We finally compute the Hamming distance of π to $(0,0,0,1,0,1,1,1)$, which is the truth table of the Boolean majority function of three inputs. Notice that a pattern distance equal to 0 does not imply that all network functions match the majority rule.



(a) Network 58 (b) Network 20

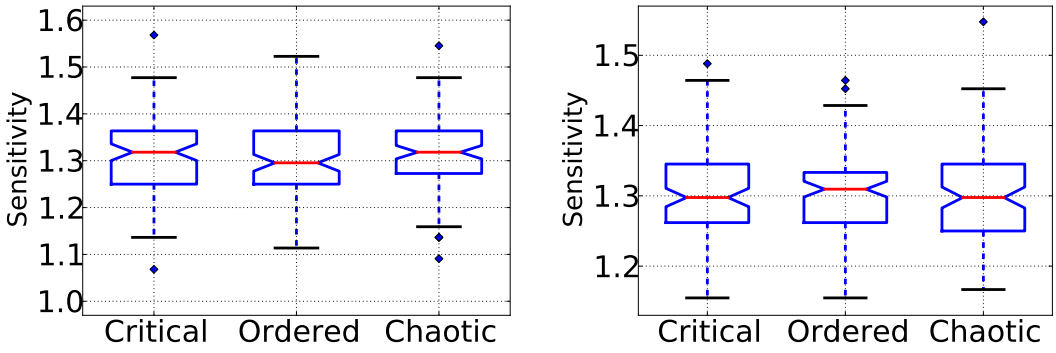
Figure 9.16: Example of overtraining on two BNs in the critical ensemble, $N = 11$. Dashed and continuous line show the classification error on the training examples and on the testing examples respectively.

Sensitivities of the optimised networks have similar characteristics regardless of the dynamical regime of the initial networks (Figure 9.17). This is a further explanation why the distributions of the classification error depicted in Figure 9.15 do no significantly differ from one another. However, the distribution of the number of attractors of optimised networks and their average periods (not shown) just slightly differ. This observation needs more detailed considerations. Optimised networks have a sensitivity of about 1.3 regardless of both initial dynamical state *and* number of nodes. Yet again, we have an experimental confirmation of the fact that networks which undergo an optimisation process could exhibit features from all dynamical classes These issues need more extended analyses and will be investigated in further work.

Figure 9.18 illustrates the evolution of the pattern distance throughout the search process. The y-axis represents the pattern distance averaged over a network ensemble and the x-axis represents the iteration index. Each data point $(i, \overline{d})$ is obtained by calculating the pattern distance d on the incumbent solution at iteration i and averaging over all networks in the ensemble, thereby obtaining $\overline{d}$ (we show only two examples, since the curves of all the other cases are substantially identical). These graphics further remark that optimised networks have similar characteristics. Specifically, they show that the search process is drawn towards networks with functions similar to the majority rule.

Comparison with a genetic algorithm

Since the search landscape of the DCP is not highly correlated, a Genetic Algorithm (GA) could attain a better performance because of its capability of sampling wide areas of the search space. Therefore, we repeated the previous experiments by plugging a GA as the optimisation component inside the training algorithm. Furthermore, by these further tests, we can also address the question as to whether the results presented in Section 9.6.2 depend on the



(a) $N = 11$ (b) $N = 21$

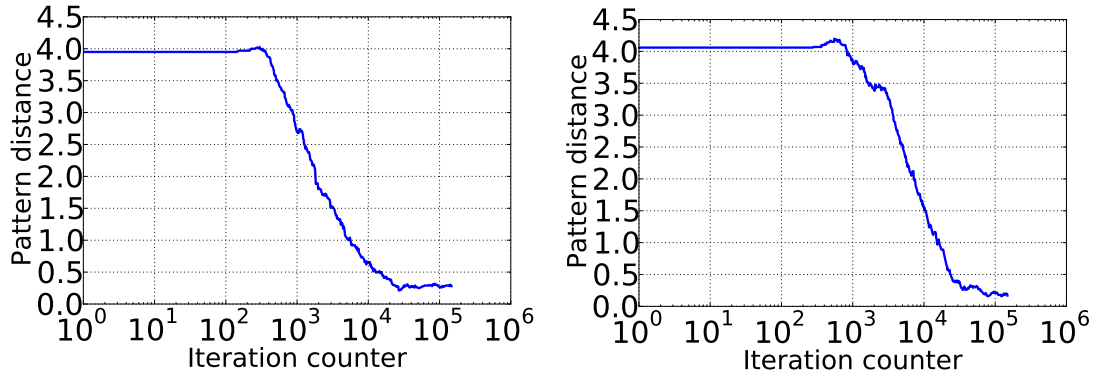
Figure 9.17: Distribution of average network sensitivity for optimised networks.

search algorithm used.

Coherently with all other experiments described in this chapter, our genetic algorithm does not modify the network topology: all solutions in the populations share the same topology, which is kept fixed throughout the search procedure. The individuals of the candidate solution population in the GA are represented by genomes, whose structure is rather similar to the one used in Section 9.2. Formally, a genome is a vector $\langle f_1, f_2, \ldots, f_N \rangle$ of N genes. A gene is a truth table $f_i \in \{0,1\}^8$ which defines the i-th node transition function. The offspring of the current population is built by generating new individuals by means of two genetic operators, i.e., crossover and mutation. The crossover operator is a standard two-parents one-point crossover. This operator is applied with probability p_{cross}. If crossover is not applied, the two parents are simply cloned. Contrarily to what we did in Section 9.2, crossover does not modify the single truth-table of a function, but, instead, it operates on the whole genome. As for mutation, we experimented with two *ad hoc* operators. The first one, labelled *X-flip*, randomly picks X bits in a truth table and negates them. The second operator, labelled *node-function*, replaces a gene with another randomly chosen K-variable Boolean function.[17] Both operations are applied to all N genes with probability p_{mut}, meaning that, on average, $N \cdot p_{mut}$ genes are changed. We implemented a steady-state genetic algorithm solver with population size of 100 genomes and roulette wheel selection. Population overlap for the steady state is 50% meaning that at each generation an offspring of 50% population size genomes is generated and added to the current population. Afterwards, the current population is shrunk back to the initial size by removing the worst genomes.

We tested several algorithm configurations which differed in mutation operator (1-flip, 2-flip and node-function), mutation probability p_{mut} and crossover probability p_{cross} for a total of 12 configurations (see Table 9.5 for a summary). The reason we chose such extreme values for p_{cross} was to verify whether

[17]This last operator works only with topologies with constant in-degree, which is the case in our experiments.



(a) Critical ensemble $N = 11$

(b) Chaotic ensemble $N = 21$

Figure 9.18: Examples of typical pattern distance trend for optimised networks. Each data point represents the pattern distance value averaged on all optimised networks in 9.18a *(a)* the critical ensemble ($N = 11$) and 9.18b *(b)* the chaotic ensemble ($N = 21$).

Name	Values
Mutation operator	1-flip, 2-flip, node-function
Mutation probability	0.1, 0.2
Crossover probability	0.1, 0.9

Table 9.5: Summary of GA parameters.

crossover introduced a too disruptive change in the network structures so as to degrade GA's performances, like we did in Section 9.2. As for mutation operators are concerned, we chose 1- and 2-flip mutations because their effect on a BN is comparable to the application of (a small number of) local search moves. On the other hand, we chose an operator such as node-function because we wanted to test the effect of a more radical modification in a BN structure. For the sake of brevity, we will refer to a configuration with a triple (mutation operator, p_{mut}, p_{cross}). Each configuration was evaluated 30 times for each dynamical class. For each algorithm evaluation, the initial population was initialised with 100 RBNs, all sharing the same topology, generated with appropriate bias depending on the dynamical class (see the beginning of Section 9.6.1). We terminated each algorithm run after 150000 objective function evaluations, the same termination condition used for ILS.

We performed a full factorial analysis of the configuration space and we subsequently applied a *Mann-Whitney test* [57], with a significance level equal to 0.05, to all pairs of algorithm configurations in order to obtain the best for each dynamical class of the initial networks. The test is able to identify two configuration, namely (node-function, 0.1, 0.9) and (node-function, 0.1, 0.1), that are significantly better than the others. This result holds true for all dynamical classes. Since these configurations are not distinguishable by the statistical test, we select the first one as the competitor against ILS.

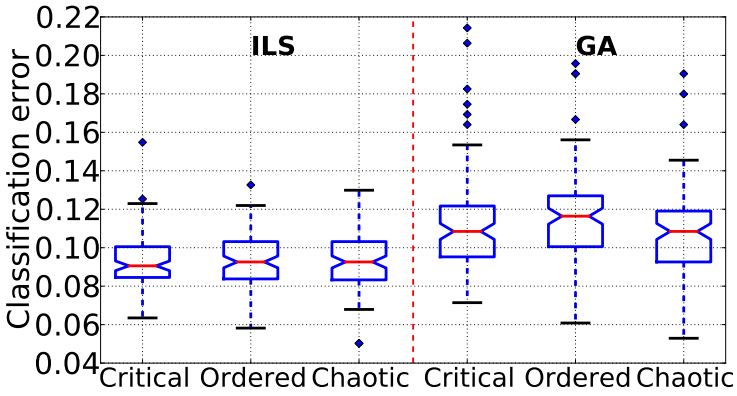


Figure 9.19: Comparison between ILS and GA performance. Graphics report the classification error distribution on the test set. Experimental setting consists of 300 RBNS with 21 nodes from each dynamical class.

Figure 9.19 compares the performance of the selected GA configuration (node-function, 0.1, 0.9) (rightmost boxplots) with our ILS (leftmost boxplots) for all dynamical classes. The comparison was performed on the same set of 300 RBNs with 21 nodes introduced in Section 9.6.1.

First of all, we observe that the GA attains results which do not qualitatively differ from those obtained by ILS, because differences among the three dynamical classes are rather small. However, the overall performance of the GA is significantly lower[18] than that of ILS, since notches of boxplot associated to ILS do not overlap to those related to the GA [48]. This result might seem counterintuitive, given the low correlation of the search landscape. Nevertheless, we should remark that the values of autocorrelation we computed span across a wide range, therefore ILS can take advantage of its fully exploration of the neighbourhood and find paths towards improving solutions which are not easily found by the sampling process of the GA.

9.7 Conclusions and discussion

BNs have been mainly considered as GRN models and are the subject of notable works in the complex systems biology literature. Nevertheless, in spite of their similarities with neural networks, their potential as learning systems has not yet been fully investigated and exploited.

In this work we use BNs as flexible objects, which can evolve by means of suitable optimisation processes, to deal with four notable issues: (*1*) the problem of generating a family of BNs with a desired attractor period; (*2*) the task of controlling the BN's trajectory to match a target state; (*3*) the automatic design of BNs with maximally distant attractors; and (*4*) the Density Classification Problem. The tasks have different characteristics which have repercussions on the search itself: in fact, the initial dynamical regime could facilitate or slow

[18] with 95% confidence

down the learning process (as in the case of the matching task), or could have no particular consequences (as in the Density Classification Problem). The optimisations starting from critical BNs do not show performances higher than those starting from different dynamical regimes. This is very likely to be a consequence of the static nature of the proposed tasks, that have not time dependent assignments: in these cases, there are no particular reasons that can favour critical systems with respect more ordered or chaotic ones. We found that the reason for the differences we observed in the performances is to be found in the autocorrelation of the search landscape, which depends on both the dynamical class and the objective function (plus the neighbourhood relation used in the local search). As a further observation, we can note that in all cases BNs successfully deal with the proposed challenges, revealing flexibility.

Further work will address the still open questions, such as: the interaction between the neighbourhood definition (that is, the allowed moves of the meta-heuristic algorithms) and the possibility of shaping the dynamical regime of these nets, by involving, not only the functions expressed by each single networks' node, but also more global network properties as, for instance, their connectivity distribution, modularity and assortativity.

Results confirm that BNs that undergo an evolutionary process exhibit a mixture of characteristics from different dynamical classes. It follows that another important open issue worthwhile to pursue is how to characterise the dynamical behaviour of ensembles of optimised networks. We believe that this could be achieved by measuring several quantities in a BN besides the ones already mentioned, such as basin entropy [135], coherency [238], Lempel-Ziv complexity [211], Normalised Compression Distance [166] and mutual information [179]. It is also possible extend known tools, like in [81], where generalised and modified Derrida plots are defined. The endeavor of finding new measures that better characterise BN can be fruitful also to the study of complex dynamical systems; in the context of ensemble approach, of particular interest are those measures not tied to a specific model, such as the last three which come from information theory, because they would enlarge its applicability.

Although control theory offers mathematical tools for steering simple engineered and natural systems towards a desired state, a framework to control complex self-organised systems is still lacking (of course the work by Barabasi [143] provides strong theoretical arguments, but so far no experimental evidence has been presented as to whether the theory of driver nodes is effective also for non-linear complex systems). BNs and the already present knowledge on their dynamical behaviour could help in this enterprise, and allow in such a way the design of specialised learning systems, able to dynamically shape their own learning capabilities in relation to the characteristics of the problem and of the search space.

Conclusions

In this thesis we described and applied a metaheuristic-based approach for automatically building models of complex biological systems. Such a methodology, which recasts our modeling problem at hand into an optimisation problem and solves it, could be also useful to design artificial systems possessing desirable properties such as robustness and evolvability. Although this methodology is not new in general, we demonstrated its successfulness in solving modeling problems in bioinformatics (Chapters 3, 4 and 6) and complex system biology (Chapter 9).

In the first part of the thesis we showed hybrid metaheuristic algorithms which constitute the current state of the art solvers for the Haplotype Inference by parsimony and the Founder Sequence Reconstruction Problem. In particular, we would like to underline the interplay of exact and approximate techniques has proved particularly effective, as shown in Chapter 6. We described different formulations of the problems and proposed several enhancements. Regarding Haplotype Inference, the new constraint programming-based approach we suggested eliminates all the drawbacks of the application of our hybrid and Master-Slave algorithms to Haplotype Inference with missing data. Moreover, Section 5.2 presents a clique finding algorithm potentially useful to compute bounds for the problem or good heuristic initial solutions for Haplotype Inference under either parsimony or minimum entropy hypotheses. An important contribution derived from the research on Haplotype Inference is also the Master-Slave approach described in Chapter 4, which proved to be useful in different contexts besides genomics. Regarding FSRP, we designed and tested several `LNS` variants and speed-ups (Section 6.8). Our `LNS` algorithm turned out to be superior to plain exact techniques—RecBlock—with the additional advantages of being *(i)* anytime solvers and; *(ii)* complete (i.e., they return a proof of optimality) if enough time is provided. Finally, we thoroughly compared our metaheuristics with the best available combinatorial algorithms to date on extensive benchmarks of real-world and simulated instances.

One of the main open question that deserves more investigation is how to characterise complexity. As shown in Chapter 7, if we limit ourselves to RBNs it is possible to define analytical measures that describe their qualitative dynamical behaviour. Specifically, we recall that computing the Lyapunov exponent of a RBN simply amounts to calculating the average network sensitivity; we also notice that computing such measure does not involve expensive simulations or calculations. If we are able to classify a RBN dynamics, theoretical studies also provide us with useful information, such as the growth of attractors with the number of nodes [66, 191] or the formation of avalanches [175]. Nevertheless, Chapter 9 presents examples about the fact that networks undergoing an evolutionary process exhibit a mixture of characteristics typical of RBNs in different

dynamical regimes. So, for instance, resulting networks from experiments in Section 9.5 are characterised by high sensitivity (typical of chaotic RBNs) and high number of attractors with short periods (typical of critical RBNs). We conjecture that this can be the result of the interplay between the search process and the objective function, although the precise causal relation still eludes us. To accomplish this task of characterising BN dynamics, many options are available, some of which are presented in Section 9.7. In particular we would like to focus the attention towards measures based on information theoretical concepts because their applicability is general and, in a future, may be used for different models in additions to BNs. This is particularly important in the context of the ensemble approach. The ensemble approach establishes that a model is an accurate description of a phenomenon on the basis of statistical properties of the real system and the model. A property definable and measurable on both the real system and its model would be indeed very useful. As a counterexample, network sensitivity provides useful information on RBN dynamics, but it is hardly definable on GRNs, i.e., on the systems we try to model with RBNs; on the contrary, a statistical property such as avalanche distribution has wider applicability and can be defined on both RBNs and GRNs [202].

Throughout the thesis, we have taken into account only deterministic problems. Although the evaluation of BN models of Chapter 9 made use of sampling, the desiderata we demanded of our networks were completely deterministic and static in nature, and the apparent stochasticity was due to technical limitations than to inherent properties of the task. A natural extension of this work would be to tackle stochastic problems, or, in the more general vision of this thesis, the automatic generation of stochastic models. For instance, any procedure that configures the parameters of an optimisation algorithm can fit in this scenario.

From a methodological point of view, one could argue that metaheuristics can be substituted by machine learning techniques, but it is important to notice that metaheuristic and machine learning approaches can be, in fact, complimentary. In an abstract scenario, a metaheuristic could learn, for example, the structure of a Bayesian network and a machine learning technique can fit its parameters to a set of samples. Moreover, pattern recognition techniques can indeed be useful for the ensemble approach, since it basically is a statistical method.

As a final remark, we would like to draw the attention to a recent trend in the field of optimisation that has many common characteristics to the methodology we exploited. The idea of "Programming by Optimisation" originates from the research on parameter configuration of stochastic algorithms and consists in applying automated procedures to build optimal solvers for the problem at hand (see [111, 117, 118, 163, 244]. The solution space searched by these "meta-solvers" is typically quite large and the relationships between its element are often non-linear. The strikingly similarities make us think that in the future this approach to design will be more and more widespread.

Proposal for future research

The research presented in this thesis can be extended in numerous ways. In this section we present some potentially fruitful continuations of the studies described in each chapter.

For what Haplotype Inference is concerned, we are going to thoroughly test our metaheuristics with other formulations of the problem, mainly those with trios and unknowns. Extension to trios should require localised modification to complementarity in order to take into account the extra familiar relationship. The case of unknowns, as we wrote at the end of Chapter 3, might be more difficult to treat, because, as opposed to trios, unknown sites add degrees of freedom and not additional constraints. Instead of tweaking our algorithms to take into account unknowns, the most convenient way is to adopt the refined model of haplotype described in Chapter 5.

The contributions in Chapter 5 can potentially be useful in a variety of cases. A future research direction will consist in studying the effectiveness of the clique finding algorithm for `HIP` and the generalised haplotype model. At first we will limit ourselves to redefine metaheuristic algorithms we proposed in this thesis, namely, the ACO in Chapter 2 and the MSG in Chapter 4. This will be done in order to determine whether the simple application of a different, more sophisticated haplotype model can be beneficial. Moreover, it is worthwhile to test if application of generalised haplotypes can improve solution quality in the case of unknowns. At a later stage, we will design *ad hoc* metaheuristic algorithms tailored to the generalised haplotype model; in particular, we aim to extend the constraint programming facility supporting this model by including trio relations. The clique finding algorithm in Section 5.2 will be the object of further experimental search. First of all, it is necessary to measure its complexity in the typical case; to do so, an appropriate benchmark should be defined, containing different kinds of instances, both real-world and simulated. Secondly, we aim to determine the effectiveness of such algorithm as heuristic guide: this algorithm guarantees to find the haplotypes with maximum coverage; this property could indeed be further exploited in an heuristic for maximum parsimony or even minimum entropy formulation. For instance, an algorithm could take the set of haplotypes with maximum coverage and, by application of Clark's Rule, construct a feasible solution; afterwards, a local search algorithm could be run to iteratively improve such solution.

We will continue to maintain and work on `EasyGenetic` library. Thanks to the advantages of generic programming, it is possible to seamlessly integrate a variety of genetic operators and population update strategies. We also consider the possibility of supporting open source technologies for parallel programming, such as OpenMP [167, 170].

Although we managed to design the state-of-the-art algorithm, the research on FSRP can be still carried on. The algorithm we proposed is modular and independent of its embedded solver. A first thing that could be easily looked into is substituting RecBlock with another *ad hoc* procedure especially tailored for solving the $k_f = 1$ and $k_f = 2$ case, as suggested in Section 6.8.5. Another important step to undertake is to test the validity of FSRP against real biological data. The long term target would be, then, to apply in sequence a Haplotype Inference algorithm followed by our FSRP solver and assess the plausibility of the produced solution. From there, one could modify either algorithm to take into account a more precise genetic model. From a more abstract point of view, it is interesting to study also different objective functions for the FSRP, in particular, the one that allows for single-site mutations [178].

We plan on continuing the work on the Boolean Network Toolkit and to extend the simulator in various directions. First of all, we will implement differ-

ent update schemes, such as random update and asynchronous update. Other scheduled extensions include support for other network models, in particular, Threshold BNs and Glass networks. Also, a more flexible interface to manipulate networks and network topologies is in the making. This interface could, for example, facilitate the construction of coupled interconnected BNs arranged in a lattice structure, reminiscent of a cellular tissue [200, 230], or forming hierarchies [152]. Of course, as our research on BNs develops, the experiments performed on the Boolean Network Toolkit will be incorporated into the source code and made available to the community.

Chapter 9 demonstrated that our method proved successful on a variety of cases. A promising line of research would be to apply our method to specific models of cellular dynamics. One of the more recent works in this regard, is the one by Serra et al. [12, 198, 228]. The main idea is that a cell type is not represented by a single attractor but, instead, by a set of attractors, which is stable under a certain level of external noise, represented by a single flip of a node. Flips are only allowed to happen when a BN has reached a stable state; after noise is applied, the network goes through a transient and relaxes into an attractor. For each attractor A_i in the state space and for all its states $s_j \in A_i$, a single bitflip is applied to every node; the attractor A_j which the BN goes to is recorded into an adjacency matrix (c^i_j), which, in the end, will contain the occurrences of all transitions $A_i \to A_j$. If we normalise every row c^i, we can interpret the resulting matrix (p^i_j)[19] as a transition matrix[20] where p^i_j represents the probability by which the BN goes from steady state A_i to steady state A_j under random external noise. (p^i_j) can be graphically represented by a weighted directed graph. We can then define a probability threshold θ inversely proportional to the noise level.[21] If we remove all edges with weight less than θ, we disconnect parts of the transition graphs. This allows us to define the key concept of Threshold Ergodic Set (TES), i.e., strongly connected components in the transition graph, which in our model represent a cell type. As we gradually increase the threshold from 0—when we have only one TES, i.e., a totipotent cell—more TESs start to appear; in particular, TESs with lower θ subdivide into TESs with higher θ. Increased cell differentiation is thus tied to changes in the threshold *theta* (and therefore in the noise level). This hierarchy of TESs can be represented by a tree.

It is of practical importance finding BNs whose hierarchy of TESs correspond to differentiation diagrams of real cells. From an experimental point of view, this is a challenging problem for two main reasons. First of all, its simulation is far from trivial: one has to sample a BN's attractor landscape, perturb the attractors and recursively construct a transition graph. Secondly, and perhaps more importantly, it is extremely difficult to define a suitable objective function, mostly because it is hard to translate in a formula such complex requirements on (hierarchies of) TESs. We recall that an objective function has the goal of guiding the algorithm in its search for the best solution; if an objective function creates a search space topology with large plateaus or a uncorrelated landscape, the search algorithm is not effective.

Section 9.6 showed that it is possible to train a Boolean network classifier.

[19] Such matrix is called right stochastic matrix.
[20] Notice the analogy with Markov Chains with finite state spaces.
[21] Intuitively, the rarer a transition the higher the noise level needed to trigger it.

In this respect, we aim to continue the investigation of the capabilities of BNs
al learning systems by tackling different classification problems.

Appendices

Appendix A

Applications of MSG Framework to Routing Problems

In this chapter we present the results attained by our MSG technique described in Chapter 4. We apply an MSG algorithm to two hard combinatorial graph-related problems, namely, the Capacitated Vehicle Routing Problem (Section A.1) and the Capacitated Minimum Spanning Tree (Section A.2). Material in both sections is taken from [14].

A.1 MSG for the Capacitated Vehicle Routing Problem

As a first non-biological case study, we show the development of a MSG algorithm for the Capacitated Vehicle Routing Problem (CVRP). This problem is highly relevant in operations research and logistics and it is known to be a hard combinatorial optimization problem [15].

A graph $G(V, E)$ is given, where V is the vertex set and E is the edge set. The vertex 0 is the depot, such as the special vertex in which an unlimited number of vehicles of capacity Q are located. The vertices $V \setminus \{0\}$ are the customers and with demands $q_i, i \in V \setminus \{0\}$. Each customer has to be visited exactly once. A routing cost $c_{ij} \geq 0$ is associated to each edge $(i, j) \in E$. We define a *route*, as the tour a vehicle performs starting from the depot and servicing a set of customers. A route is feasible if the sum of the customer demands does not exceed the vehicle capacity Q. The CVRP consists of determining a set of feasible routes with minimum total routing cost, in which the demand of each customer is satisfied.

Our master-slave algorithm is a generalization of the Clarke-Wright saving heuristic first proposed in [55] and further refined in [169]. The main components of a Clarke-Wright heuristic are a list of pairs of customers, called *merge list*, and a deterministic constructive procedure which, starting from an infeasible disaggregated solution made of one route for each client node, iteratively picks the first pair of customers from the list and possibly merges two routes if some

feasibility conditions are satisfied. The key idea is thus to order the saving list according to a heuristic function, called *saving function*, on the pairs of customers. The saving function is crucial because it determines the ordering of the elements in the merge list.

It is easy now to recognize the main ingredients for a master-slave algorithm. From this perspective, a merge list is but an input to a deterministic slave procedure which returns a set of routes.

In [169] has been proposed a saving function which takes into account three different components weighted by three coefficients whose values are taken in the cube $[0, 2]^3$. An good, but possibly not optimal, configuration has been found through repeated evaluation of the algorithm. In a similar vein, our MSG takes a population of merge lists and, by exploiting the adaptive search of genetic algorithms, iteration after iteration produces "better" merge lists. Of course, in this context, "better" means "merge lists with higher fitness", that is, merge lists that, when provided as inputs to the slave procedure, return low-cost solutions.

A.1.1 Proposed algorithm and evaluation

In this section we describe the best MSG algorithm instance for the CVRP, labelled SPLIT.200.9.25%, out of the 36 possible variants studied in [14] after extensive parameter tuning.

Following the guideline outlined in 4.3.2, we describe the characteristics of SPLIT.200.9.25% starting with chromosome structure. In this algorithm an individual should represent the ordering of the merge list, therefore, a chromosome is structurally a permutation of indices. By applying a chromosome/permutation to a *reference merge list*, computed once at the beginning of the algorithm and kept constant throughout the search, we obtain an ordered merge list to be fed to the slave. The reference merge list is ordered using the saving function proposed in [55]. The initial population is composed by 10% identical permutations and 90% random permutations in order to provide adequate diversification in the population and guarantee that a fraction of initial chromosomes has good fitness. Population has constant size of 200 individuals and every iteration an offspring of 200 more chromosomes is generated. We implemented an *ad-hoc* variation of the recombination operators for permutations described in Section 4.3.2: crossover and mutation are performed separately on the head of the chromosome (i.e., 25% of the whole) and on the tail (i.e., the remaining 75% of the whole). Crossover is always applied; mutation operator performs, on average, $m_r = 9$ swaps within head and tail. Roulette-wheel was chosen as selection strategy and population update is steady-state.

In order to test the performance of our MSG algorithm, we considered the same test instances as in [169], that consist of a well known benchmark set for the CVRP. Table A.1 reports the computational results obtained on each test set (namely *A*, *B*, *P*, *E* and *CMT*), by SPLIT.200.9.25% and a randomized multistart algorithm (MULTISTART-CVRP). The randomized multistart algorithm consists of a multistart implementation of the algorithm by Öncan et al. [169], in which the parameters of the saving function are randomly selected within the interval $[0, 5]$. The randomized multistart algorithm is iterated as long as the time limit is reached. SPLIT.200.9.25% was run 5 times considering two time limits: 1 minute and 5 minutes for each instance, for a total runtime bud-

Table A.1: CVRP experimental results. Entries are the average percentage deviations from the best known solutions.

	5 minutes				25 minutes			
	SPLIT.200.9.25%		MULTISTART-CVRP		SPLIT.200.9.25%		MULTISTART-CVRP	
A	0.286	(0.448)	2.048	(1.150)	0.278	(0.431)	2.032	(1.160)
B	0.292	(0.659)	1.746	(1.051)	0.288	(0.660)	1.733	(1.037)
P	0.411	(0.421)	2.910	(2.290)	0.411	(0.421)	2.896	(2.288)
E	0.521	(0.625)	2.113	(1.837)	0.432	(0.620)	1.777	(1.445)
CMT	1.641	(1.498)	3.541	(2.138)	1.362	(1.246)	3.392	(2.172)
Avg.	0.630	(0.730)	2.472	(1.693)	0.554	(0.676)	2.366	(1.620)

get of 5 and 25 minutes respectively. In order to provide a fair comparison, MULTISTART-CVRP was allotted the same time limits, i.e., 5 and 25 minutes.

Although the multistart algorithm does not represent the state-of-the-art, this comparison is useful to empirically measure the effectiveness and contribution of the master algorithm to the search. This comparison assesses the effectiveness of the genetic learning mechanism with respect to a randomized search. Results show that MSG effectively enhances a saving heuristic better than a randomized search and that the solutions returned are closer to the best known results. For each algorithm and computing time, the first column in the table (denoted by the name of the adopted algorithm) report the average percentage deviation with respect to the best known solution values in the literature. The second column reports in parenthesis the standard deviation. The last line of the table reports the column averages.

A.2 MSG for the Capacitated Minimum Spanning Tree

The second non biological problem studied, the Capacitated Minimum Spanning Tree (CMSTP) [172], is closely related to the CVRP both in its definition and its resolution strategy. Let us consider an undirected graph $G = (V, E)$, where $V = \{0, \ldots, n\}$ is the vertex set and $E = \{(i, j) \mid i, j \in V\}$ is the edge set. Vertex 0 is the *root*, whereas the remaining vertices $i \in V \setminus \{0\}$ have a non negative demands d_i. Each edge $(i, j) \in E$ is associated with a non-negative cost c_{ij} and the edges incident with the root are called *gates*. The CMSTP calls for the determination of a minimum cost spanning tree such that the demand served on each subtree linked to the root through a gate does not exceed a given maximum capacity Q.

Likewise the CVRP, a constructive heuristic is available. The Esau-Williams algorithm (EW) is the first method proposed in the literature to solve the CMSTP and it is a constructive saving-based heuristic essentially identical to the well-known Clarke and Wright algorithm mentioned in Section A.1: both algorithms, in fact, receive as input an merge list ordered according some saving function. The only difference between the two is the definition of merge action, since the CMSTP requires to build a set of trees and not cycles.

A.2.1 Proposed algorithm and evaluation

The proposed algorithm belongs to the same family of MSG as SPLIT.200.9.25%. Like in the previous case, we describe only the best MSG algorithm configuration, obtained after comparing 36 algorithm instances (see [14] for a statistical proof), which we call PARAMS.200.9.3. The algorithm is remarkably similar to the one described in Section A.1.1 with some differences outlined in the following. Chromosome structure is again a permutation of indices, but we implement the permutation-specific recombination operators described in Section 4.3.2 instead of custom variants as we did for the CVRP. Crossover probability is 1.0 and mutation rate is 9.

The biggest difference is the population initialization method. Instead of generating random permutations, we rely on an effective heuristic to generate orderings of the reference merge list. We consider the three-parameter saving function by Öncan et al. [168]: for every initial chromosome, we randomly choose a triplet of parameter in the cube $[0, 3]^3$ and compute the saving values of each merge in the merge list; afterwards we sort the merge list according the saving value; finally, by comparing the sorted merge list to the reference one, we obtain a permutation of indices which is our initial chromosome. Population size is again constantly set to 200. Each generation 200 new individual are generated.

Algorithm evaluation was carried out on the same set of instances as in [168], namely *tc40*, *te40*, *tc80*, *te80*, *cm50*, *cm100* and *cm200*, where the number denotes maximum capacity Q and the letters denote topological properties of instance graphs. PARAMS.200.9.3 is compared once again to a random multistart implementation, MULTISTART-CMST, which repeatedly runs the heuristic by Öncan [168], each time with random parameters uniformly sampled in the cube $[0, 5]^3$. PARAMS.200.9.3 was run 5 times each instance against two time limits of 1 and 5 minutes; totally, it spent 5, or 25, minutes on each instance. In order to provide fair evaluation, MULTISTART-CMST was executed for 5 and 25 minutes per instance. Table A.2 reports the average percentage deviations from the best known solutions (in parentheses) and the average standard deviations across all instance sets. Moreover, the last line of the table displays the column averages.

A.3 Conclusions and discussion

We have presented a master-slave approach to three different hard combinatorial problems: Haplotype Inference by Maximum Parsimony, Capacitated Vehicle Routing and Capacitated Minimum Spanning Tree problems. The master searches in the space of parameters used by the slave to build a solution. We have implemented the master as a genetic algorithm and the slave as an adaptation of a deterministic constructive procedure readily available from the literature. Results show that the approach is very efficient and reaches a good balance between solution quality and running time.

The approach is general and the algorithm can be extended in several directions. First of all, different procedures for implementing the slave can be adopted, especially stochastic ones, even though they require a more complex solution quality estimation. Besides extending the slave, also the algorithm implemented by the master can be changed. For example, other population-

Table A.2: CMSTP computational results. Parenthesised entries are the average percentage deviations from the best known solutions.

	5 minutes				25 minutes			
	Params.200.9.3		Multistart-cmst		Params.200.9.3		Multistart-cmst	
tc40Q3	0.105	(0.151)	1.384	(0.614)	0.028	(0.056)	1.384	(0.614)
tc40Q5	0.643	(0.464)	1.527	(0.271)	0.643	(0.464)	1.527	(0.271)
tc40Q10	0.000	(0.000)	0.320	(0.640)	0.000	(0.000)	0.320	(0.640)
tc80Q5	3.381	(0.808)	4.164	(0.855)	2.667	(0.852)	4.001	(0.610)
tc80Q10	1.963	(0.999)	2.959	(0.919)	1.387	(0.884)	2.890	(0.836)
tc80Q20	0.710	(0.620)	0.766	(0.969)	0.71	(0.620)	0.766	(0.969)
te40Q3	-0.036	(0.072)	0.780	(0.516)	-0.036	(0.072)	0.780	(0.516)
te40Q5	0.318	(0.399)	2.354	(0.755)	0.217	(0.434)	2.354	(0.755)
te40Q10	0.584	(0.657)	2.628	(1.709)	0.584	(0.657)	2.628	(1.709)
te80Q5	0.771	(0.400)	1.303	(0.564)	0.544	(0.371)	1.248	(0.544)
te80Q10	2.578	(0.501)	3.764	(0.825)	2.161	(0.229)	3.715	(0.864)
te80Q20	1.093	(1.224)	3.711	(2.418)	0.790	(1.116)	3.549	(2.356)
cm50Q200	0.567	(0.633)	1.819	(1.272)	0.546	(0.650)	1.819	(1.272)
cm50Q400	1.271	(0.835)	2.862	(0.888)	1.138	(0.778)	2.862	(0.888)
cm50Q800	1.883	(0.903)	2.997	(1.510)	1.605	(0.840)	2.997	(1.510)
cm100Q200	14.888	(1.814)	20.043	(2.424)	11.918	(1.764)	19.438	(3.068)
cm100Q400	6.017	(1.291)	8.908	(1.744)	5.297	(1.315)	7.649	(0.806)
cm100Q800	1.101	(0.480)	1.977	(1.170)	0.986	(0.522)	1.759	(0.868)
cm200Q200	27.005	(2.485)	25.441	(3.056)	22.36	(1.675)	24.863	(2.716)
cm200Q400	20.475	(4.415)	21.281	(4.272)	15.173	(3.454)	19.062	(3.803)
cm200Q800	8.837	(2.573)	8.453	(3.450)	4.756	(1.758)	7.274	(2.769)
Avg.	4.484	(1.034)	5.688	(1.469)	3.499	(0.881)	5.375	(1.352)

based metaheuristics can be chosen, such as Ant colony optimization [65] or also stochastic local search techniques [110] can be tested. Furthermore, the approach could also be improved by adding a mechanism such as the one used in Benders decomposition techniques [109], in which the slave return the master a set of constraints to reduce the search space.

Bibliography

[1] Adcock, S. (2011). GAUL: Genetic algorithm utility library. `http://gaul.sourceforge.net/`. Viewed: November 2011.

[2] Ahuja, R., Ergun, Ö., Orlin, J., and Punnen, A. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102.

[3] Albert, I., Thakar, J., Li, S., Zhang, R., and Albert, R. (2008). Boolean network simulations for life scientists. *Source Code for Biology and Medicine*, 3(1):16.

[4] Aldana, M., Balleza, E., Kauffman, S., and Resendiz, O. (2007). Robustness and evolvability in genetic regulatory networks. *Journal of Theoretical Biology*, 245:433–448.

[5] Aldana, M., Coppersmith, S., and Kadanoff, L. (2003). Boolean dynamics with random couplings. In Kaplan, E., Marsden, J., and Sreenivasan, K., editors, *Perspectives and Problems in Nonlinear Science. A celebratory volume in honor of Lawrence Sirovich*, Springer Applied Mathematical Sciences Series, pages 23–90. Springer, Heidelberg, Germany.

[6] Ansaloni, L., Villani, M., and Serra, R. (2009). Dynamical critical systems for information processing: a preliminary study. In [229].

[7] Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156.

[8] Bafna, V. and Bansal, V. (2004). The number of recombination events in a sample history: Conflict graph and lower bounds. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:78–90.

[9] Balleza, E., Alvarez-Buylla, E., Chaos, A., Kauffman, S., Shmulevich, I., and Aldana, M. (2008a). Critical dynamics in genetic regulatory networks: Examples from four kingdoms. *PLoS ONE*, 3(6):e2456.

[10] Balleza, E., Alvarez-Buylla, E., Chaos, A., Kauffman, S., Shmulevich, I., and Aldana, M. (2008b). Critical dynamics in genetic regulatory networks: Examples from four kingdoms. *PLoS ONE*, 3(6):e2456.

[11] Bar–Yam, Y. (1997). *Dynamics of Complex Systems*. Studies in nonlinearity. Addison–Wesley, Reading, MA.

[12] Barbieri, A., Villani, M., Serra, R., Kauffman, S., and Colacci, A. (2009). Extended notion of attractors in noisy random Boolean networks. In [229].

[13] Bastolla, U. and Parisi, G. (1996). Closing probabilities in the Kauffman model: An annealed computation. *Physica D*, 98:1–25.

[14] Battarra, M., Benedettini, S., and Roli, A. (2011). Leveraging saving-based algorithms by master-slave genetic algorithms. *Engineering Applications of Artificial Intelligence*, 24:555–566.

[15] Battarra, M., Golden, B., and Vigo, D. (2008). Tuning a parametric Clarke–Wright heuristic via a genetic algorithm. *Journal of the Operations Research Society*, 59:1568–1572.

[16] Benedettini, S. (2011a). The Boolean Network Toolkit. Available at: `http://booleannetwork.sourceforge.net`. Viewed: November 2011.

[17] Benedettini, S. (2011b). EasyGenetic. Viewed: 11-Nov-2011.

[18] Benedettini, S., Blum, C., and Roli, A. (2010). A randomized iterated greedy algorithm for the founder sequence reconstruction problem. In Blum, C. and Battiti, R., editors, *Proceedings of the Fourth Learning and Intelligent OptimizatioN Conference – LION 4*, volume 6073 of *Lecture Notes in Computer Science*, pages 37–51. Springer, Heidelberg, Germany.

[19] Benedettini, S., Di Gaspero, L., and Roli, A. (2008a). Towards a highly scalable hybrid metaheuristic for haplotype inference under parsimony. *Hybrid Intelligent Systems, International Conference on*, pages 702–707.

[20] Benedettini, S., Di Gaspero, L., and Roli, A. (2009a). Genetic master-slave algorithm for haplotype inference by parsimony. Technical Report DEIS-LIA-09-003, University of Bologna (Italy). LIA Series no. 93.

[21] Benedettini, S., Roli, A., and Di Gaspero, L. (2009b). EasyGenetic: A template metaprogramming framework for genetic master-slave algorithms. In Stützle, T., Birattari, M., and Hoos, H., editors, *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics*, volume 5752 of *Lecture Notes in Computer Science*, pages 135–139. Springer Berlin / Heidelberg.

[22] Benedettini, S., Roli, A., and Di Gaspero, L. (2009c). Easygenetic: A template metaprogramming framework for genetic master-slave algorithms. Technical Report DEIS-LIA-09-005, University of Bologna (Italy). LIA Series no. 95.

[23] Benedettini, S., Roli, A., and Gaspero, L. (2008b). Two-level ACO for haplotype inference under pure parsimony. In *Proceedings of the 6th international conference on Ant Colony Optimization and Swarm Intelligence*, ANTS '08, pages 179–190. Springer Berlin / Heidelberg.

[24] Benedettini, S., Roli, A., Serra, R., and Villani, M. (2011). Stochastic local search to automatically design Boolean networks with maximally distant attractors. In Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A., Merelo, J., Neri, F., Preuss, M., Richter, H., Togelius,

J., and Yannakakis, G., editors, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 22–31. Springer, Heidelberg, Germany.

[25] Bernasconi, A. and Codenotti, B. (1993). Sensitivity of Boolean functions, harmonic analysis, and circuit complexity. Technical report, International Computer Science Institute, Berkley, CA.

[26] Bertolazzi, P., Godi, A., Labbé, M., and Tininini, L. (2008). Solving haplotyping inference parsimony problem using a new basic polynomial formulation. *Comput. Math. Appl.*, 55(5):900–911.

[27] Birattari, M. (2009). *Tuning Metaheuristics: A Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence*. Springer, Heidelberg, Germany.

[28] Birattari, M., Balaprakash, P., Stützle, T., and Dorigo, M. (2008). Estimation-based local search for stochastic combinatorial optimization using delta evaluations: A case study on the probabilistic traveling salesman problem. *INFORMS Journal on Computing*, 20(4).

[29] Blin, G., Rizzi, R., Sikora, F., and Vialette, S. (2011). Minimum mosaic inference of a set of recombinants. In Potanin, A. and Viglas, T., editors, *Proceedings of 17th Computing: the Australasian Theory Symposium – CATS 2011*, CRPIT.

[30] Blum, C., Blesa, M., Roli, A., and Sampels, M., editors (2008). *Hybrid Metaheuristics: An Emerging Approach to Optimization*, volume 114 of *Studies in Computational Intelligence*. Springer, Heidelberg, Germany.

[31] Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308.

[32] Blum, C.and Blesa Aguilera, M. J. and Roli, A.and Sampels, M., editors (2008). *Hybrid Metaheuristics – An Emerging Approach to Optimization*, volume 114 of *Studies in Computational Intelligence*. Springer.

[33] bnsim (2011). BNSim. `http://code.google.com/p/bnsim/`. Viewed: November 2011.

[34] Bongard, J. C. (2011). Spontaneous evolution of structural modularity in robot neural network controllers. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 251–258, New York, NY, USA. ACM.

[35] boost (2011). Boost `C++` libraries. `http://www.boost.org/`. Viewed: November 2011.

[36] Brent, R. P. (1980). An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20:176–184.

[37] Brown, D. and Harrower, I. (2004). A new integer programming formulation for the pure parsimony problem in haplotype analysis. In Jonassen, I. and Kim, J., editors, *Algorithms in Bioinformatics*, volume 3240 of *Lecture Notes in Computer Science*, pages 254–265. Springer Berlin / Heidelberg.

[38] Brown, D. G. and Harrower, I. M. (2006). Integer programming approaches to haplotype inference by pure parsimony. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(2):141–154.

[39] Browning, B. L. and Browning, S. R. (2009). A unified approach to genotype imputation and haplotype-phase inference for large data sets of trios and unrelated individuals. *The American Journal of Human Genetics*, 84(2):210–223.

[40] Browning, S. R. and Browning, B. L. (2011). Haplotype phasing: existing methods and new developments. *Nature Reviews Genetics*, 12(10):703–714.

[41] Cahon, S., Melab, N., and Talbi, E.-G. (2004). ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380.

[42] Carchrae, T. and Beck, J. (2005). Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387.

[43] Caseau, Y. and Laburthe, F. (1999). Effective forget-and-extend heuristics for scheduling problems. In Focacci, F., Lodi, A., Milano, M., and Vigo, D., editors, *Electronic Proceedings of the Int. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 1999*.

[44] Catanzaro, D., Godi, A., and Labbé, M. (2010). A class representative model for pure parsimony haplotyping. *INFORMS Journal on Computing*, 22:195–209.

[45] Catanzaro, D. and Labbé, M. (2009). The pure parsimony haplotyping problem: overview and computational advances. *International Transactions in Operational Research*, 16(5):561–584.

[46] Chambers, J. (1983). *Graphical Methods for Data Analysis*. Springer, Berlin, Germany.

[47] Chambers, J., Cleveland, W., Kleiner, B., and Tukey, P. (1983). *Graphical Methods for Data Analysis*. Wadsworth & Brooks/Cole.

[48] Chambers, J. M. and Cleveland, W.S. and Kleiner, B. and Tukey, P.A. (1983). *Graphical Methods for Data Analysis*. Chapman and Hall, New York.

[49] Chau, K. W. (2004). A two-stage dynamic model on allocation of construction facilities with genetic algorithm. *Automation in Construction*, 13:481–490.

[50] Cheng, D. and Qi, H. (2009). Controllability and observability of boolean control networks. *Automatica*, 45(7):1659–1667.

[51] Cheng, D., Qi, H., and Li, Z. (2011). *Analysis and Control of Boolean Networks: A Semi-tensor Product Approach*. Communications and Control Engineering. Springer.

[52] Chiarandini, M., Dumitrescu, I., and Stützle, T. (2008). Very large-scale neighborhood search: Overview and case studies on coloring problems. In [30], pages 117–150.

[53] Chiarandini, M. and Stützle, T. (2002). An application of iterated local search to graph coloring problem. In Johnson, D. S., Mehrotra, A., and Trick, M., editors, *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125.

[54] Clark, A. G. (1990). Inference of haplotypes from PCR-amplified samples of diploid populations. *Molecular Biology and Evolution*, 7:111–122.

[55] Clarke, G. and Wright, J. W. (1964). Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12:568–581.

[56] Climer, S., Jäger, G., Templeton, A. R., and Zhang, W. (2009). How frugal is mother nature with haplotypes? *Bioinformatics*, 25(1):68–74.

[57] Conover, W. (1999). *Practical Nonparametric Statistics*. John Wiley & Sons, Hoboken, NJ, USA, 3rd edition.

[58] Daly, M. J., Rioux, J. D., Schaffner, S. F., Hudson, T. J., and Lander, E. S. (2001). High-resolution haplotype structure in the human genome. *Nature Genetics*, 29:229–232.

[59] Derrida, B. and Pomeau, Y. (1986). Random networks of automata: a simple annealed approximation. *Europhysics Letters*, 1(2):45–49.

[60] Di Gaspero, L. and Roli, A. (2008). Stochastic local search for large-scale instances of the haplotype inference problem by pure parsimony. *Journal of Algorithms: Algorithms in Logic, Informatics and Cognition*, 63(1–3):55–69.

[61] Di Gaspero, L. and Roli, A. (2009). Flexible stochastic local search for haplotype inference. In Selman, B., Battiti, R., and Stützle, T., editors, *Learning and Intelligent Optimization - Third International Conference, LION 2009*, volume 5851 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin / Heidelberg.

[62] Di Gaspero, L. and Schaerf, A. (2003). EasyLocal++: An object-oriented framework for flexible design of local search algorithms. *Software: Practice and Experience*, 33(8):733–765.

[63] Doman, C. (2008). Boolean Network Modeler. `http://www.rustyspigot.com/software/BooleanNetwork/`. Viewed: November 2011.

[64] Dorigo, M. (1994). Learning by probabilistic Boolean networks. In *Proceedings of World Congress on Computational Intelligence – IEEE International Conference on Neural Networks*, pages 887–891, Orlando, Florida.

[65] Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. MIT Press, Cambridge, MA, USA.

[66] Drossel, B. (2005). Number of attractors in random Boolean networks. *Physcal Review E*, 72:016110.

[67] ducktype (2011). Duck typing Wikipedia page. Viewed: 11-Nov-2011.

[68] EClab (2011). Viewed: 11-Nov-2011.

[69] Eén, N. and Sörensson, N. (2006). Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26.

[70] El-Araby, E. E., Yorino, N., and Sasaki, H. (2003). A two level hybrid ga/slp for facts allocation problem considering voltage security. *International Journal of Electrical Power & Energy Systems*, 25:327–335.

[71] El-Mabrouk, N. and Labuda, D. (2004). Haplotypes histories as pathways of recombinations. *Bioinformatics*, 20(12):1836–1841.

[72] Erdem, E. and Türe, F. (2008). Efficient haplotype inference with answer set programming. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, pages 436–441. AAAI Press.

[73] Eronen, L., Geerts, F., and Toivonen, H. (2004). A markov chain approach to reconstruction of long haplotypes. In *Pacific Symposium on Biocomputing. World Scientific*, pages 104–115.

[74] Eronen, L., Geerts, F., and Toivonen, H. (2006). Haplorec: efficient and accurate large-scale reconstruction of haplotypes. *BMC Bioinformatics*, 7(1):542.

[75] Esau, L. R. and Williams, K. C. (1966). On teleprocessing system design. *IBM Systems Journal*, 5:142–147.

[76] Esmaeili, A. and Jacob, C. (2008). Evolution of discrete gene regulatory models. In Keijzer, M., editor, *Proceedings of GECCO'08 – Genetic and Evolutionary Computation Conference*, pages 307–314, Atlanta, GA.

[77] Espinosa-Soto, C. and Wagner, A. (2010). Specialization can drive the evolution of modularity. *PLoS Comput Biol*, 6(3):e1000719+.

[78] Fates, N. (2011). Stochastic cellular automata solve the density classification problem with an arbitrary precision. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science – STACS 2011*, Leibniz International Proceedings in Informatics (LIPIcs), pages 284–295.

[79] Fiedler, M. (1973). Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(2):298–305.

[80] Fretter, C. and Drossel, B. (2008). Response of Boolean networks to perturbations. *European Physical Journal B*, 62:365–371.

[81] Fretter, C., Szejka, A., and Drossel, B. (2009). Perturbation propagation in random and evolved Boolean networks. *New Journal of Physics*, 11(3):033005:1–13.

[82] Freund, Y. and Schapire, R. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.

[83] Frigge, M., Hoaglin, D., and Iglewicz, B. (1989a). Some implementations of the boxplot. *The American Statistician*, 43(1):50–54.

[84] Frigge, M., Hoaglin, D., and Iglewicz, B. (1989b). Some implementations of the boxplot. *The American Statistician*, 43(1):50–54.

[85] Gamma, E., Johnson, R., Helm, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[86] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[87] Georgopoulou, C. A. and Giannakoglou, K. C. (2009). Two-level, two-objective evolutionary algorithms for solving unit commitment problems. *Applied Energy*, 86:1229–1239.

[88] Gershenson, C. (2004). Introduction to random Boolean networks. In Bedau, M., Husbands, P., Hutton, T., Kumar, S., and Suzuki, H., editors, *Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (ALife IX)*, pages 160–173, Boston, MA.

[89] Gershenson, C. (2011). RBNLab. `http://sourceforge.net/projects/rbn/`. Viewed: November 2011.

[90] Glass, L. and Hill, C. (1998). Ordered and disordered dynamics in random networks. *Europhysics Letters*, 41(6):599–604.

[91] Glover, F. W. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA.

[92] Goldberg, D. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison–Wesley, Reading, MA.

[93] Graça, A., Lynce, I., Marques-Silva, J. a., and Oliveira, A. L. (2010). Haplotype Inference by Pure Parsimony: A Survey. *Journal of Computational Biology*, 17(8):969–992.

[94] Graça, A., Marques-Silva, J., Lynce, I., and Oliveira, A. L. (2007). Efficient haplotype inference with pseudo-boolean optimization. In Anai, H., Horimoto, K., and Kutsia, T., editors, *Algebraic Biology, Second International Conference, AB 2007, Castle of Hagenberg, Austria, July 2-4, 2007, Proceedings*, volume 4545 of *Lecture Notes in Computer Science*, pages 125–139. Springer Berlin / Heidelberg, Berlin-Heidelberg, Germany.

[95] Graça, A., Marques-Silva, J., Lynce, I., and Oliveira, A. L. (2011). Haplotype inference with pseudo-boolean optimization. *Annals of Operations Research*, 184:137–162.

[96] Graça, A., Marques-Silva, J. a., Lynce, I., and Oliveira, A. L. (2008). Efficient haplotype inference with combined cp and or techniques. In Perron, L. and Trick, M., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5015 of *Lecture Notes in Computer Science*, pages 308–312. Springer Berlin / Heidelberg.

[97] Graudenzi, A. and Serra, R. (2008). A new model of genetic network: the gene protein boolean network. In Serra, R., Villani, M., and Poli, I., editors, *Artificial life and evolutionary computation – Proceedings of WIVACE 2008*, pages 283–291. World Scientific Publishing, Singapore.

[98] Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A. (2006). Concepts: linguistic support for generic programming in c++. *SIGPLAN Not.*, 41:291–310.

[99] Guerri, A. and Milano, M. (2004). Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, (ECAI 2004)*, pages 475–479. IOS Press.

[100] Gusev, A., Pasaniuc, B., and Mandoiu, I. (2008). Highly scalable genotype phasing by entropy minimization. *IEEE/ACM Trans. on Computational Biology and Bioinformatics*, 5(2):252–261.

[101] Gusfield, D. (2001). Inference of haplotypes from sample diploid populations: complexity and algorithms. *Journal of Computational Biology*, 8:305–323.

[102] Gusfield, D. (2003). Haplotype inference by pure parsimony. In Baeza-Yates, R. A., Chávez, E., and Crochemore, M., editors, *Combinatorial Pattern Matching (CPM 2003), Proceedings of the 14th Annual Symposium*, volume 2676 of *Lecture Notes in Computer Science*, pages 144–155, Berlin-Heidelberg, Germany. Springer Berlin / Heidelberg.

[103] Halldórsson, B., Bafna, V., Edwards, N., Lippert, R., Yooseph, S., and Istrail, S. (2004). A survey of computational methods for determining haplotypes. In *Lecture Notes in Computer Science (2983): Computational Methods for SNPs and Haplotype Inference*, pages 26–47. Springer.

[104] Halldórsson, B. V., Bafna, V., Edwards, N., Lippert, R., Yooseph, S., and Istrail, S. (2002). A survey of computational methods for determining haplotypes. In Istrail, S., Waterman, M. S., and Clark, A. G., editors, *Computational Methods for SNPs and Haplotype Inference*, volume 2983 of *Lecture Notes in Computer Science*, pages 26–47, Berlin-Heidelberg, Germany. Springer Berlin / Heidelberg.

[105] Halperin, E. and Karp, R. M. (2005). The minimum-entropy set cover problem. *Theoretical Computer Science*, 348:240–250.

[106] Hansen, P. and Mladenović, N. (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467.

[107] Heckel, R., Schober, S., and Bossert, M. (2010). On random boolean threshold networks. In *Source and Channel Coding (SCC), 2010 International ITG Conference on*, pages 1–6.

[108] Holland, J. (1975). *Adaption in natural and artificial systems.* The University of Michigan Press, Ann Harbor, MI.

[109] Hooker, J. (2007). *Integrated Methods for Optimization.* Operations Research & Management Science. Springer.

[110] Hoos, H. and Stützle, T. (2005). *Stochastic Local Search Foundations and Applications.* Morgan Kaufmann Publishers, San Francisco, CA (USA).

[111] Hoos, H. H. (2009). Programming by optimisation: Computer-aided design of high-performance algorithms.

[112] Hordijk, W. (1996). A measure of landscapes. *Evolutionary Computation*, 4:335–360.

[113] Huang, S. and Ingber, D. (2006,2007). A non-genetic basis for cancer progression and metastasis: Self-organizing attractors in cell regulatory networks. *Breast Disease*, 26:27–54.

[114] Huang, Y. T., Chao, K. M., and Chen, T. (2005). An approximation algorithm for haplotype inference by maximum parsimony. *Journal of Computational Biology*, 12(10):1261–1274.

[115] Hudson, R. (2002). Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics*, 18:337–338.

[116] Hudson, R. and Kaplan, N. (1985). Statistical properties of the number of recombination events in the history of a sample of dna sequences. *Genetics*, 111:147–164.

[117] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2010). Automated configuration of mixed integer programming solvers. In Lodi, A., Milano, M., and Toth, P., editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 186–202. Springer.

[118] Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stẗzle, T. (2009). ParamILS: An automatic algorithm configuration framework. *J. Artif. Intell. Res. (JAIR)*, 36:267–306.

[119] Iliadis, A., Watkinson, J., Anastassiou, D., and Wang, X. (2010). A haplotype inference algorithm for trios based on deterministic sampling. *BMC Genetics*, 11(1):78.

[120] Istvan, A. (2011). BooleanNet. `http://code.google.com/p/booleannet/`. Viewed: November 2011.

[121] Jager, G., Climer, S., and Zhang, W. (2009). Complete parsimony haplotype inference problem and algorithms. volume 5757 of *Lecture Notes in Computer Science*, pages 337–348. Springer Berlin / Heidelberg.

[122] Jain, A., Murty, M., and Flynn, P. (1999). Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323.

[123] Kaneko, K. (2006). *Life: An Introduction to Complex System Biology*. Springer, Berlin, Germany.

[124] Kappler, K., Edwards, R., and Glass, L. (2003). Dynamics in high-dimensional model gene networks. *Signal Processing*, 83:789–798.

[125] Kauffman, S. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22:437–467.

[126] Kauffman, S. (1986). Adaptive automata based on Darwinian selection. *Physica D*, 22:68–82.

[127] Kauffman, S. (1993). *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, UK.

[128] Kauffman, S. (2004). A proposal for using the ensemble approach to understand genetic regulatory networks. *Journal of Theoretical Biology*, 230:581–590.

[129] Kesseli, J., Rämö, P., and Yli-Harja, O. (2005). On spectral techniques in analysis of Boolean networks. *Physica D: Nonlinear Phenomena*, 206(1-2):49–61.

[130] Khajavirad, A., Michalek, J., and Simpson, T. (2009). An efficient decomposed multiobjective genetic algorithm for solving the joint product platform selection and product family design problem with generalized commonality. *Structural and Multidisciplinary Optimization*, 39:187–201.

[131] Kimmel, G. and Shamir, R. (2005). A block-free hidden markov model for genotypes and its application to disease association. *Journal of Computational Biology*, 12:1243–1260.

[132] Kirkpartick, S., Gelatt, C., and Vecchi, M. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

[133] Knuth, D. (1998). *The Art Of Computer Programming, Volume 2: Seminumerical Algorithms, 3/E*. Pearson Education.

[134] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, 1 edition.

[135] Krawitz, P. and Shmulevich, I. (2007). Basin entropy in Boolean network ensembles. *Phys. Rev. Lett.*, 98(15):158701:1–4.

[136] Lacomme, P., Prins, C., and Ramdane-Cherif, W. (2005). Evolutionary algorithms for periodic arc routing problems. *European Journal of Operational Research*, 165:535–553.

[137] Lancia, G., Pinotti, M. C., and Rizzi, R. (2004). Haplotyping populations by pure parsimony: Complexity of exact and approximation algorithms. *INFORMS Journal on Computing*, 16(4):348–359.

[138] Land, M. and Belew, R. K. (1995). No perfect two-state cellular automata for density classification exists. *Physical Review Letters*, 74(25):5148–5150.

[139] Langton, C. G. (1989). *Artificial life: the proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, held September, 1987, in Los Alamos, New Mexico.* Santa Fé Institute studies in the sciences of complexity. Addison-Wesley.

[140] Lemke, N., Mombach, J., and Bodmann, B. (2001). A numerical investigation of adaptation in populations of random Boolean networks. *Physica A*, 301:589–600.

[141] Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2006). *Empirical Hardness Models for Combinatorial Auctions*, chapter 19, pages 479–504. MIT Press, Cambridge, MA, USA.

[142] Liang, K. and Wang, X. (2008). A deterministic sequential monte carlo method for haplotype inference. *IEEE Journal of Selected Topics in Signal Processing*, 2:322–331.

[143] Liu, Y.-Y., Slotine, J.-J., and Barabasi, A.-L. (2011). Controllability of complex networks. *Nature*, 473(7346):167–173.

[144] Lourenço, H., Martin, O., and Stützle, T. (2003). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 320–353. Springer, New York, NY, New York, NY.

[145] Luque, B. and Solé, R. (2000). Lyapunov exponents in random Boolean networks. *Physica A-statistical Mechanics and Its Applications*, 284:33–45.

[146] Lynce, I. and Marques-Silva, J. (2006a). Efficient haplotype inference with boolean satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, Menlo Park, CA, USA. AAAI Press.

[147] Lynce, I. and Marques-Silva, J. (2006b). SAT in bioinformatics: Making the case with haplotype inference. In Biere, A. and Gomes, C. P., editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 136–141. Springer Berlin / Heidelberg.

[148] Lynce, I., Marques-Silva, J. a., and Prestwich, S. (2008). Boosting haplotype inference with local search. *Constraints*, 13:155–179.

[149] Lyngsø R. and Song, Y. (2005). Minimum recombination histories by branch and bound. In Casadio, R. and Myers, G., editors, *Proceedings of the 5th Workshop on Algorithms in Bioinformatics – WABI 2005*, volume 3692 of *Lecture Notes in Computer Science*, pages 239–250. Springer, Heidelberg, Germany.

[150] Manfroni, M. (2011). Towards Boolean network design for robotics applications. Master's thesis, University of Bologna, Second Faculty of Engineering.

[151] Marchini, J., Cutler, D., Patterson, N., Stephens, M., Eskin, E., Halperin, E., Lin, S., Qin, Z. S., Munro, H. M., Abecasis, G. R., Donnelly, P., and International HapMap Consortium (2006). A comparison of phasing algorithms for trios and unrelated individuals. *American Journal of Human Genetics*, 78:437–450.

[152] Markert, E. K., Baas, N., Levine, A. J., and Vazquez, A. (2010). Higher order Boolean networks as models of cell state dynamics. *Journal of Theoretical Biology*, 264(3):945–951.

[153] Marques-Silva, J., Lynce, I., Graça, A., and Oliveira, A. L. (2007). Efficient and tight upper bounds for haplotype inference by pure parsimony using delayed haplotype selection. In Neves, J., Santos, M., and Machado, J., editors, *Progress in Artificial Intelligence*, volume 4874 of *Lecture Notes in Computer Science*, pages 621–632. Springer Berlin / Heidelberg.

[154] Mesot, B. and Teuscher, C. (2005). Deducing local rules for solving global tasks with random Boolean networks. *Physica D*, 211:88–106.

[155] Meyer, B. (2008). Hybrids of constructive metaheuristics and constraint programming: A case study with aco. In Blum, C., Aguilera, M., Roli, A., and Sampels, M., editors, *Hybrid Metaheuristics*, volume 114 of *Studies in Computational Intelligence*, pages 151–183. Springer Berlin / Heidelberg.

[156] Meyer, B. and Ernst, A. (2004). Integrating ACO and constraint propagation. In Dorigo, M., Birattari, M., Blum, C., Luca, Mondada, F., and Stützle, T., editors, *Ant Colony, Optimization and Swarm Intelligence: 4th International Workshop, ANTS 2004*, volume 3172 of *Lecture Notes in Computer Science*, pages 166–177. Springer Verlag GmbH.

[157] Michałewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Heidelberg, Germany.

[158] Mihaljev, T. and Drossel, B. (2009). Evolution of a population of random Boolean networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 67:259–267.

[159] Müssel, C., Hopfensitz, M., and Kestler, H. A. (2010). BoolNet–an R package for generation, reconstruction and analysis of Boolean networks. *Bioinformatics (Oxford, England)*, 26(10):1378–1380.

[160] Müssel, C., Hopfensitz, M., and Kestler, H. A. (2011). BoolNet. `http://www.informatik.uni-ulm.de/ni/mitarbeiter/HKestler/boolnet/`. Viewed: November 2011.

[161] Myers, S. and Griffiths, R. (2003). Bounds on the minimum number of recombination events in a sample history. *Genetics*, 163(1):375–394.

[162] Neigenfind, J., Gyetvai, G., Basekow, R., Diehl, S., Achenbach, U., Gebhardt, C., Selbig, J., and Kersten, B. (2008). Haplotype inference from unphased snp data in heterozygous polyploids based on sat. *BMC Genomics*, 9:1–26.

[163] Nell, C., Fawcett, C., Hoos, H. H., and Leyton-Brown, K. (2011). HAL: A framework for the automated analysis and design of high-performance algorithms. In Coello Coello, C. A., editor, *LION*, volume 6683 of *Lecture Notes in Computer Science*, pages 600–615. Springer.

[164] Niu, T., Qin, Z. S., Xu, X., and Liu, J. S. (2002). Bayesian haplotype inference for multiple linked single-nucleotide polymorphisms. *The American Journal of Human Genetics*, 70(1):157–169.

[165] Nolfi, S. and Floreano, D. (2000). *Evolutionary robotics*. The MIT Press, Cambridge, MA.

[166] Nykter, M., Price, N., Aldana, M., Ramsey, S., Kauffman, S., Hood, L., Yli-Harja, O., and Shmulevich, I. (2008). Gene expression dynamics in the macrophage exhibit criticality. In *Proceedings of the National Academy of Sciences, USA*, volume 105, pages 1897–1900.

[167] omp (2011). The openmp api specification for parallel programming. Viewed: November 2011.

[168] Öncan, T. and Altınel, I. K. (2009). Parametric enhancements of the Esau-Williams heuristic for the capacitated minimum spanning tree problem. *Journal of the Operational Research Society*, 60:259–267.

[169] Öncan, T. and Altınel, K. (2005). A new enhancement of the Clarke and Wright savings heuristic for the capacitated vehicle routing problem. *Journal of the Operational Research Society*, 56:954–961.

[170] OpenMP Architecture Review Board (2008). Openmp application program interface. Specification.

[171] Packard, N. (1988). Adaptation toward the edge of chaos. In Kelso, J., Mandell, A., and Shlesinger, M., editors, *Dynamic Patterns in Complex Systems*, pages 293–301. World Scientific, Singapore.

[172] Papadimitriou, C. (1978). The complexity of the capacitated tree problem. *Networks*, 8:217–230.

[173] Patarnello, S. and Carnevali, P. (1986). Learning networks of neuron with Boolean logic. *Europhysics Letters*, 4(4):503–508.

[174] Perron, L., Shaw, P., and Furnon, V. (2004). Propagation guided large neighborhood search. In Wallace, M., editor, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 468–481. Springer, Heidelberg, Germany.

[175] Rämö, P., Kesseli, J., and Yli-Harja, O. (2006). Perturbation avalanches and criticality in gene regulatory networks. *Journal of Theoretical Biology*, 242(1):164–170.

[176] Rastas, P., Koivisto, M., Mannila, H., and Ukkonen, E. (2005). A hidden markov technique for haplotype reconstruction. In *WABI. Volume 3692 of Lecture Notes in Computer Science*, pages 140–151. Springer.

[177] Rastas, P., Kollin, J., and Koivisto, M. (2008). Fast bayesian haplotype inference via context tree weighting. In *Proceedings of the 8th international workshop on Algorithms in Bioinformatics*, WABI '08, pages 259–270, Berlin, Heidelberg. Springer Berlin / Heidelberg.

[178] Rastas, P. and Ukkonen, E. (2007). Haplotype inference via hierarchical genotype parsing. In Giancarlo, R. and Hannenhalli, S., editors, *Proceedings of the 7th Workshop on Algorithms in Bioinformatics – WABI2007*, volume 4645 of *Lecture Notes in Computer Science*, pages 85–97. Springer, Heidelberg, Germany.

[179] Ribeiro, A., Kauffman, S., Lloyd-Price, J., Samuelsson, B., and Socolar, J. (2008). Mutual information in random Boolean models of regulatory networks. *Physical Review E*, 77:011901:1–10.

[180] Rohlf, T. and Bornholdt, S. (2002). Criticality in random threshold networks: annealed approximation and beyond. *Physica A: Statistical Mechanics and its Applications*, 310(1-2):245–259.

[181] Roli, A., Arcaroli, C., Lazzarini, M., and Benedettini, S. (2009a). Boolean networks design by genetic algorithms. In [229], page 13.

[182] Roli, A., Benedettini, S., Serra, R., and Villani, M. (2011a). Analysis of attractor distances in Random Boolean networks. In Apolloni, B., Bassis, S., Esposito, A., and Morabito, C., editors, *Neural Nets WIRN10 – Proceedings of the 20th Italian Workshop on Neural Nets*, volume 226 of *Frontiers in Artificial Intelligence and Applications*, pages 201–208. IOS Press.

[183] Roli, A., Benedettini, S., Stützle, T., and Blum, C. (2010). Additional material to the paper 'Large Neighbourhood Search Algorithms for the Founder Sequences Reconstruction Problem'. Available at `www.lia.deis.unibo.it/~aro/research/fsrp/`.

[184] Roli, A., Benedettini, S., Stützle, T., and Blum, C. (2012). Large neighbourhood search algorithms for the founder sequence reconstruction problem. *Computers & Operations Research*, 39(2):213–224.

[185] Roli, A. and Blum, C. (2009). Tabu search for the founder sequence reconstruction problem: A preliminary study. In Omatu, S., Rocha, M., Bravo, J., Fernández-Riverola, F., Corchado, E., Bustillo, A., and Corchado, J., editors, *Proceedings of the 3rd International Workshop on Practical Applications of Computational Biology and Bioinformatics – IWPACBB'09*, volume 5518 of *Lecture Notes in Computer Science*, pages 1035–1042. Springer, Heidelberg, Germany.

[186] Roli, A., Manfroni, M., Pinciroli, C., and Birattari, M. (2011b). On the design of Boolean network robots. In Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A., Merelo, J., Neri, F., Preuss, M., Richter, H., Togelius, J., and Yannakakis, G., editors, *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 43–52. Springer, Heidelberg, Germany.

[187] Roli, A., Serra, R., and Benedettini, S. (2009b). Clustering di attrattori di reti Booleane casuali. In *Modelli, sistemi e applicazioni di Vita Artificiale e Computazione Evolutiva – WIVACE 2009*, pages 167–176. Fridericiana editrice.

[188] Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049.

[189] Ruiz, R. and Stützle, T. (2008). An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. *European Journal of Operational Research*, 187(3):1143–1159.

[190] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 3rd edition.

[191] Samuelsson, B. and Troein, C. (2005). Random maps and attractors in random boolean networks. *Physcal Review E*, 72:046112.

[192] Santiago-Mozos, R., Salcedo-Sanz, S., DePrado-Cumplido, M., and Bousoño-Calzón, C. (2005). A two-phase heuristic evolutionary algorithm for personalizing course timetables: a case study in a spanish university. *Computers & Operations Research*, 32:1761–1776.

[193] Scheet, P. and Stephens, M. (2006). A fast and flexible statistical model for large-scale population genotype data: Applications to inferring missing genotypes and haplotypic phase. *The American Journal of Human Genetics*, 78(4):629–644.

[194] Schilstra, M., Wegner, K., and Block, Marceland Egri-Nagy, A. (2011). NetBuilder'. `http://strc.herts.ac.uk/bio/maria/Apostrophe/index.htm`. Viewed: November 2011.

[195] Schwartz, R., Clark, A., and Istrail, S. (2002). Methods for inferring blockwise ancestral history from haploid sequences. In Guigó, R. and Gusfield, D., editors, *Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 44–59. Springer, Heidelberg, Germany.

[196] Schwarzer, C. (2003). Random Boolean Network Toolbox. `http://www.mathworks.com/matlabcentral/fileexchange/3231`. Viewed: November 2011.

[197] Serra, R. and Villani, M. (2002). Perturbing the regular topology of cellular automata: implications for the dynamics. In Chopard, B., Tomassini, M., and Bandini, S., editors, *Cellular Automata, 5th International Conference on Cellular Automata for Research and Industry – ACRI 2002*, volume 2493 of *Lecture Notes in Computer Science*, pages 168–177. Springer, Heidelberg, Germany.

[198] Serra, R., Villani, M., Barbieri, A., Kauffman, S., and Colacci, A. (2010a). On the dynamics of random Boolean networks subject to noise: attractors, ergodic sets and cell types. *Journal of Theoretical Biology*, 265(2):185–193.

[199] Serra, R., Villani, M., Barbieri, A., Kauffman, S., and Colacci, A. (2010b). On the dynamics of random Boolean networks subject to noise: Attractors, ergodic sets and cell types. *Journal of Theoretical Biology*, 265(2):185–193.

[200] Serra, R., Villani, M., Damiani, C., Graudenzi, A., and Colacci, A. (2008). The diffusion of perturbations in a model of coupled random boolean networks. In Umeo, H., Morishita, S., Nishinari, K., Komatsuzaki, T., and Bandini, S., editors, *Cellular Automata*, volume 5191 of *Lecture Notes in Computer Science*, pages 315–322. Springer Berlin / Heidelberg.

[201] Serra, R., Villani, M., Graudenzi, A., and Kauffman, S. (2007). Why a simple model of genetic regulatory networks describes the distribution of avalanches in gene expression data. *Journal of Theoretical Biology*, 246:449–460.

[202] Serra, R., Villani, M., and Semeria, A. (2004a). Genetic network models and statistical properties of gene expression data in knock-out experiments. *Journal of Theoretical Biology*, 227:149–157.

[203] Serra, R., Villani, M., and Semeria, A. (2004b). Genetic network models and statistical properties of gene expression data in knock-out experiments. *Journal of Theoretical Biology*, 227:149–157.

[204] Serra, R. and Zanarini, G. (1990). *Complex Systems and Cognitive Processes*. Springer, Heidelberg, Germany.

[205] SGI (2011). Standard Template Library programmer's guide. `http://www.sgi.com/tech/stl/stl_introduction.html`. Viewed: November 2011.

[206] Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In Maher, M. and Puget, J.-F., editors, *Principle and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, Heidelberg, Germany.

[207] Shestak, V., Chong, E. K. P., Siegel, H. J., Maciejewski, A. A., Benmohamed, L., Wang, I.-J., and Daley, R. (2008). A hybrid branch-and-bound and evolutionary approach for allocating strings of applications to heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 68:410–426.

[208] Shmulevich, I. and Dougherty, E. (2009). *Probabilistic Boolean Networks: The Modeling and Control of Gene Regulatory Networks*. SIAM, Philadelphia, PA.

[209] Shmulevich, I., Dougherty, E. R., Kim, S., and Zhang, W. (2002). Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274.

[210] Shmulevich, I. and Kauffman, S. (2004). Activities and sensitivities in Boolean network models. *Physical Review Letters*, 93(4):048701:1–10.

[211] Shmulevich, I., Kauffman, S., and Aldana, M. (2005). Eukaryotic cells are dynamically ordered or critical but not chaotic. *Proceedings of the National Academy of Sciences of the United States of America*, 102(38):13439–13444.

[212] Siek, J. G., Lee, L.-Q., and Lumsdaine, A. (2002). *The Boost Graph Library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[213] Song, Y., Wu, Y., and Gusfield, D. (2005). Efficient computation of close lower and upper bounds on the minimum number of recombinations in biological sequence evolution. *Bioinformatics*, 21(suppl_1):413–422.

[214] sourceforge (2011). SourceForge code repository. Online; accessed 11-Nov-2011.

[215] spgal (2007). spGAL. Viewed: 11-Nov-2011.

[216] Stephens, M. and Donnelly, P. (2003). A comparison of bayesian methods for haplotype reconstruction from population genotype data. *The American Journal of Human Genetics*, 73(5):1162–1169.

[217] Stephens, M. and Scheet, P. (2005). Accounting for decay of linkage disequilibrium in haplotype inference and missing-data imputation. *The American Journal of Human Genetics*, 76(3):449–462.

[218] Stephens, M., Smith, N. J., and Donnelly, P. (2001). A new statistical method for haplotype reconstruction from population data. *The American Journal of Human Genetics*, 68(4):978–989.

[219] Stützle, T. (2006). Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539.

[220] Stützle, T., López-Ibáñez, M., Dorigo, M., Cochran, J. J., Cox, L. A., Keskinocak, P., Kharoufeh, J. P., and Smith, J. C. (2011). *A Concise Overview of Applications of Ant Colony Optimization*. John Wiley & Sons, Inc.

[221] Szejka, A. and Drossel, B. (2007). Evolution of canalizing Boolean networks. *European Physical Journal B*, 56:373–380.

[222] Szejka, A. and Drossel, B. (2010). Evolution of Boolean networks under selection for a robust response to external inputs yields an extensive neutral space. *Phys. Rev. E*, 81(2):021908:1–9.

[223] The International HapMap Consortium (2003). The international HapMap project. *Nature*, 426:789–796.

[224] The International HapMap Consortium (2005). A haplotype map of the human genome. *Nature*, 437:1299–1320.

[225] Thyson, G., Chapman, J., Hugenholtz, P., Allen, E., Ram, R., Richardson, P., Solovyev, V., Rubin, E., Rokhsar, D., and Banfield, J. (2004). Community structure and metabolism through reconstruction of microbial genomes from the environment. *Nature*, 428:37–43.

[226] Tomassini, M., Giacobini, M., and Darabos, C. (2004). Evolution of small-world networks of automata for computation. In Yao, X., Burke, E., Lozano, J., Smith, J., Merelo-Guervós, J., Bullinaria, J., Rowe, J., Tino, P., Kabán, A., and Schwefel, H.-P., editors, *Proceedings of Parallel ProblemSolving from Nature – PPSN 2004*, volume 3242 of *Lecture Notes in Computer Science*, pages 672–681. Springer, Heidelberg, Germany.

[227] Ukkonen, E. (2002). Finding founder sequences from a set of recombinants. In Guigó, R. and Gusfield, D., editors, *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics – WABI 2002*, volume 2452 of *Lecture Notes in Computer Science*, pages 277–286. Springer, Heidelberg, Germany.

[228] Villani, M., Barbieri, A., and Serra, R. (2011). A dynamical model of genetic networks for cell differentiation. *PLoS ONE*, 6(3):e17703.

[229] Villani, M. and Cagnoni, S., editors (2009). *Proceedings of CEEI 2009 - Workshop on complexity, evolution and emergent intelligence*, Reggio Emilia, Italy.

[230] Villani, M., Serra, R., Ingrami, P., and Kauffman, S. (2006). Coupled random boolean network forming an artificial tissue. In El Yacoubi, S., Chopard, B., and Bandini, S., editors, *Cellular Automata*, volume 4173 of *Lecture Notes in Computer Science*, pages 548–556. Springer Berlin / Heidelberg.

[231] Wall, M. (2011). GAlib – A C++ Library of Genetic Algorithm Components. Viewed: 11-Nov-2011.

[232] Wang, I.-L. and Yang, H.-E. (2011). Haplotyping populations by pure parsimony based on compatible genotypes and greedy heuristics. *Applied Mathematics and Computation*, 217(23):9798–9809.

[233] Wang, R.-S., Zhang, X.-S., and Sheng, L. (2005). Haplotype inference by pure parsimony via genetic algorithm. In Zhang, X.-S., Liu, D.-G., and Wu, L.-Y., editors, *Operations Research and Its Applications: the Fifth International Symposium (ISORA'05), Tibet, China, August 8–13*, volume 5 of *Lecture Notes in Operations Research*, pages 308–318. Beijing World Publishing Corporation, Beijing, People Republic of China.

[234] Watts, D. (1999). *Small worlds: the dynamics of networks between order and randomness*. Princeton University Press, Princeton, NJ.

[235] Wegner, K., Robinson, M., Egri-Nagy, A., Knabe, J., Nehaniv, C., and Schilstra, M. (2006). NetBuilder. `http://strc.herts.ac.uk/bio/maria/NetBuilder/index.html`. Viewed: November 2011.

[236] Wikipedia (2011a). Cycle detection. `http://en.wikipedia.org/wiki/Cycle_detection`. Viewed: November 2011.

[237] Wikipedia (2011b). Generator (computer programming). `http://en.wikipedia.org/wiki/Generator_%28computer_programming%29`. Viewed: November 2011.

[238] Willadsen, K., Triesch, J., and Wiles, J. (2008). Understanding robustness in random Boolean networks. In Bullock, S., Noble, J., Watson, R., and Bedau, M. A., editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 694–701, Cambridge, MA. MIT Press.

[239] Wu, Y. (2009). An analytical upper bound on the minimum number of recombinations in the history of SNP sequences in populations. *Information Processing Letters*, 109(9):427–431.

[240] Wu, Y. and Gusfield, D. (2008). Improved algorithms for inferring the minimum mosaic of a set of recombinants. In Ma, B. and Zhang, K., editors, *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching – CPM 2007*, volume 4580 of *Lecture Notes in Computer Science*, pages 150–161. Springer, Heidelberg, Germany.

[241] Wuensche, A. (2011a). DDLab. `http://www.informatics.sussex.ac.uk/users/andywu/ddlab.html`. Viewed: November 2011.

[242] Wuensche, A. (2011b). *Exploring Discrete Dynamics*. Luniver Press.

[243] Xing, E. P., Jordan, M. I., and Sharan, R. (2007). Bayesian haplotype inference via the dirichlet process. *Journal of Computational Biology*, 14(3):267–284.

[244] Xu, L., Hoos, H., and Leyton-Brown, K. (2010). Hydra: Automatically configuring algorithms for portfolio-based selection. In Fox, M. and Poole, D., editors, *AAAI*. AAAI Press.

[245] Yang, J., Zhang, M., He, B., and Yang, C. (2009). Bi-level programming model and hybrid genetic algorithm for flow interception problem with customer choice. *Computers & Mathematics with Applications*, 57:1985–1994.

[246] Zhang, B. and Horvath, S. (2005). A general framework for weighted gene co-expression network analysis. *Statistical Applications in Genetics and Molecular Biology*, 4(1).

[247] Zhang, J.-H., Wu, L.-Y., Chen, J., and Zhang, X.-S. (2008). A fast haplotype inference method for large population genotype data. *Computational Statistics & Data Analysis*, 52(11):4891–4902.

[248] Zhang, Q., Wang, W., McMillan, L., De Villena, F.-M., and Threadgill, D. (2009). Inferring genome-wide mosaic structure. *Bioinformatics*, pages 150–161.

[249] Zlochin, M., Birattari, M., Meuleau, N., and Dorigo, M. (2004). Model-based search for combinatorial optimization: A critical survey. *Annals of Operations Research*, 131(1–4):373–395.