Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN

INGEGNERIA BIOMEDICA, ELETTRICA E DEI SISTEMI

Ciclo 34

Settore Concorsuale: 01/A6 - RICERCA OPERATIVA

Settore Scientifico Disciplinare: MAT/09 - RICERCA OPERATIVA

NETWORKS: A STUDY IN ANALYSIS AND DESIGN

Presentata da: Naga Venkata Chaitanya Gudapati

Coordinatore Dottorato

Michele Monaci

Supervisore

Enrico Malaguti

Co-supervisore

Michele Monaci

Esame finale anno 2022

Acknowledgments

I would like to sincerely thank my advisers Prof. Enrico Malaguti and Prof. Michele Monaci for being exceptional mentors for the past three years. They have been extremely kind and guided me through my research. I learnt a lot from them and they were there for me during both good and tough times.

I also want to thank my previous Advisor Prof. Tamas Terlaky for instilling in me, the curiosity for mathematical optimization. I also want to thank Prof. Antonio Frangioni at University of Pisa where I had spent three months working on Network Design. I also want to thank Dr. Utz Uwe-Haus and Dr. Alfio Lazzaro for guiding me though my secondment at CRAY Switzerland. This research was supported by "Mixed-Integer Non Linear Optimisation: Algorithms and Application" consortium, which has received funding from the European Union's EU Framework Programme for Research and Innovation Horizon 2020 under the Marie Skłodowska-Curie Actions Grant Agreement No 764759.

I also want to thank my colleagues Luca Accorsi, Federico Naldini, Henri Lefebvre, Benedikt Nguyen and Paolo Paranuzzi for being the smartest walls I could bounce my ideas off of. I would also like to I want to thank the members of "Foletti Di Montagna", especially Marco, Lelluz, Alessandro, Matthew, Margo, Giulia, Lory, Francesco, and Constantin for welcoming a complete stranger into their arms and taking me around to visit the most beautiful places in Italy. I would also like to express my immense gratitude for the staff of DEI for helping me wade through the initial days in Italy.

Abstract

In this dissertation, we will look at two fundamental aspects of Networks — Network Analysis and Network Design. In part A, we look at Network Analysis area of the dissertation which involves finding the densest subgraph in a given graph. The densest subgraph extraction problem is fundamentally a non-linear optimization problem. Nevertheless, it can be solved in polynomial time by an exact algorithm based on the iterative solution of a series of maximum flow sub-problems. Despite its polynomial time complexity, the computing time required by the exact algorithms on very large graphs could be prohibitive. Thus, to approach graphs with millions of vertices and edges, one has to resort to heuristic algorithms. We provide an efficient implementation of a greedy heuristic from the literature that is extremely fast and has some nice theoretical properties. We also introduce a new heuristic algorithm that is built on top of the greedy and the exact methods. An extensive computational analysis shows that the proposed heuristic algorithm proved very effective on a large number of test instances, often providing either the optimal solution or near-optimal solution within short computing times.

In part B, we discuss Network design which is a cornerstone of mathematical optimization, is about defining the main characteristics of a network satisfying requirements on connectivity, capacity, and level-of-service. In multi-commodity network design, one is required to design a network minimizing the installation cost of its arcs and the operational cost to serve a set of point-to-point connections. The definition of this prototypical problem was recently enriched by additional constraints imposing that each origin-destination of a connection is served by a single path satisfying one or more level-of-service requirements, thus

defining the *Network Design with Service Requirements*. These constraints are crucial, e.g., in telecommunications and computer networks, in order to ensure reliable and low-latency communication. In this paper we provide a new formulation for the problem, where variables are associated with paths satisfying the end-to-end service requirements. We present a fast algorithm for enumerating all the exponentially-many feasible paths and, when this is not viable, we provide a column generation scheme that is embedded into a branch-and-cut-and-price algorithm.

Contents

Acknowledgments **iii**

Abstract **iv**

List of Figures **viii**

List of Tables **ix**

1 Networks: A concise introduction **1**
1.1 Definitions 2
1.2 Graph Analytics: 3
1.3 Network Design 4
1.3.1 Models for Network Design 4

2 Dense Subgraphs: Introduction **6**
2.1 Definition of the problem 8

3 Algorithms to find dense subgraphs **13**
3.0.1 Exact algorithms 13
3.0.2 *Greedy Peeling* algorithm 15
3.1 *Hybrid* algorithm 20

4 Dense Subgraphs: Computational experiments **24**
4.1 Setup and programming 24
4.2 Testbed 25
4.3 Analysis on instances in the *Medium* bucket 26

4.4 Tuning of the algorithm 28
4.4.1 Disconnected graphs. 30
4.5 Results on instances in the *Large* and *Massive* buckets . . . 31
4.6 Results on weighted instances 32

5 Introduction to NDSR 38

6 Problem description and formulation of NDSR 42
6.0.1 Arc-flow formulation 43
6.0.2 Path-based formulation 44

7 Algorithms to solve NDSR 46
7.1 Variable enumeration: 46
7.1.1 Models comparison 47
7.2 Branch-and-cut-and-price approach 50
7.2.1 Column generation and labelling 50
7.2.2 Branching scheme 53
7.2.3 Adding valid inequalities 53

8 Computational experiments 56
8.1 Instances from the literature 56
8.2 Results on the instances from the literature 58
8.3 Results on additional instances 60
8.4 Strengthening the model 62

9 Conclusions and Future Work 65
9.1 Dense Subgraphs 65
9.2 NDSR 66
9.2.1 Application of NDSR to Hop Constrained Survivable Network 66

Bibliography 68

List of Figures

2.1 A small example to understand in which a clique is not the densest subgraph 11

3.1 *Goldberg's* algorithm 14
3.2 Augmented network $A(g)$, courtesy of [Gol84] 15
3.3 *Greedy Peeling* algorithm for the unweighted case. 16
3.4 Bad connected instance for the *Greedy Peeling*. The graph has $2k + t + 2p$ vertices and $4k + t + 2$ edges. 19
3.5 *Hybrid* algorithm. 20
3.6 *Expansion* phase. 21
3.7 *Expansion* phase example 23

7.1 Simple example for which the path-based formulation dominates the arc-flow formulation. 49
7.2 Fractional solution of the linear relaxation of the path-based formulation. 50

List of Tables

4.1 Results on instances in the *Medium* bucket. All times are in milliseconds. 34
4.2 Instances for which the *Hybrid* algorithm can take a very long time. 35
4.3 Average computing time and percentage gap for different variants of the *Hybrid* algorithm. All times are in milliseconds. 35
4.4 Average computing time and percentage gap for `H1` and *Hybrid* algorithm on a selected subset of instances. All times are in milliseconds. 35
4.5 Performance of *Greedy Peeling* and *Hybrid* algorithm on disconnected graphs. All times are in milliseconds. 36
4.6 Results on instances in the *Large* bucket. All times are in milliseconds. 36
4.7 Results on instances in the *Massive* bucket. All times are in milliseconds. 37
4.8 Results on weighted instances. All times are in milliseconds. 37

8.1 Results on instances from the literature. 59
8.2 Results on additional instances. 61
8.3 Results on the addition of valid inequalities. 63

1 Networks: A concise introduction

Studying *Networks* has been an integral part of the curriculum for any operation researcher. Networks, or sometimes also referred to as Graphs, are mathematical structures that can be used to represent interaction between objects. Graphs have been used for centuries to model real-world scenarios, like Euler describing Seven Bridges of Königsberg in 1736. In recent decades, tremendous advancements were made in analysing graphs both from theoretical and computational point of views. With these new advancements, it also became easier to represent various real-world scenarios as graphs. Networks are corner stones of some of the massive inventions in the past few decades. A network at its core has two components — nodes or sometimes referred to as vertices and edges or arcs, depending on the nature of the connections between the nodes. The PageRank algorithm that made Google a powerhouse in web search activity works on a graph where web pages are nodes and the hyperlinks are the arcs. Similarly, GPS and online maps which uses the road networks as graphs have changed the face of transportation industry. The interdisciplinary nature of graph theory can be shown by highlighting the works in the areas of social science, particularly in social network analysis, friend network analysis and also in linguistics. There are many other areas described in chapter 2.

1.1 Definitions

An undirected graph G is a mathematical structure that comprises of an ordered pair (V, E) where,

- V is a set of nodes or vertices
- E is a set of edges where $E \in \{\{a, b\}\} \mid a, b \in V$ and $a \;! = b$. The unordered pair of vertices $\{a, b\}$ is called an edge between the vertices a and b.

The above graph G is assumed to have a unique edge between any two vertices. An *undirected multigraph* will have multiple edges connecting the two same vertices. The graphs can also have *loops* which is an edge from a vertex to itself.

The edges in simple graphs can also have real "weights". The weights can be used to represent real world phenomenon like distance, time or importance of the connection between the two nodes. In this thesis, we deal only with either weighted or unweighted simple graphs.

A directed graph G' is similar to G but the connections between the vertices are represented by arcs which are ordered pair of nodes. G' is an ordered pair of (V, A), where,

- V is a set of nodes or vertices
- A is a set of arcs where $A \in \{(a, b)\} \mid (a, b) \in V^2$ and $a \;! = b$.

The ordered pair of vertices $\{a, b\}$ where $a! = b$ is called a directed arc between from the vertex a to vertex b. We primarily deal with simple weighted directed graphs in this thesis.

While Graphs have been used to solve innumerable real-world problems, we focus on two special cases in this thesis — Graph Analytics analysis and Network Design

1.2 Graph Analytics:

[HI18] further shows the numerous interdisciplinary fields where graph theory and analytics are indispensable. Graph analytics' primary goal is to extraction information that can be then used to solve some real world business cases. [HI18] lists various kinds of analytics that can be run on Graphs:

- **Path Analytics**: These analytics are probably the most fundamental analysis that can be done on graphs. They primarily involve graph traversal by using breadth first search (BFS) or Depth First Search (DFS) algorithms and shortest path problems using numerous algorithms like Dijkstra and Floyd-Warshall algorithms. We use these algorithms a lot in various applications throughout this paper.
- **Connectivity:** Connectivity is again a fundamental graph analytics tool and is used primarily to find connected components which in turn have lot of practical implications like finding largest components or number of connected components.
- **Ranking:** In any real world network, not all vertices are assumed to have equal importance. There are various vertex or edge centrality measures proposed in [Fre78] that highlight the importance of the vertices to the graph network. As described before, algorithms like PageRank also tell us how important a web page is in the internet.
- **Clustering and Subgraph Detection**: Clustering is one of the most studied "big-data" algorithm to find subgraphs in a given graph containing vertices with similar characteristics. We can cluster based on some rules (like cliques) or detecting communities (vertices in a community have high connectivity with other vertices and low connectivity with vertices outside the community). One can also be interested in detecting a specific subgraph that satisfies a particular metric like say density. We discuss more about finding densest subgraphs in Chapter 2

While there are other graph analytics methods that we have not mentioned, we described the most important ones related to this manuscript. For more information, readers can refer to [Bol98] and [RKF12]

1.3 Network Design

Over the past few decades, tremendous amount of research has been done in the area of Network Design to create cutting edge transportation and supply chain networks which lead to invaluable amount of savings in time, resources and money. While the area of Network Design was primarily used for transportation science problems, in the recent areas, communication networks like telecommunication and wireless communication networks are also using network design principles to effectively transfer data. [MW84] describe how network design can achieve both long-term and short-term goals of effective transportation plans. Network planning that involves building airports, shipyards, highways probably require billions of dollars in investment have to be very carefully planned and Network Design applications and algorithms plays a fundamental role in catering to the multiple constraints of resources, budget , service etc and providing with the best outcome. [MW84] also give examples for intermediate level network design like warehouse of facility location problems. Many e-commerce sites like amazon and postal carriers like UPS, FedEx have to do network planning on almost daily basis to find optimal routes for their drivers to delver the goods. To summarize, irrespective of the time frame of outcomes or scale of investment, Network Design plays a crucial role in our lives and poor network choices can waste lot of money and result in sub-optimal outcomes.

1.3.1 Models for Network Design

[MW84] mentions a generic network design problem. It consists of N, a set of vertices and A, a set of arcs. In most cases, we also have K, a set

of commodities and their respective sources and sinks, and fixed and variable or operating costs for the arcs. The network design problem tries to route these commodities from their sources to sinks trying to achieve the required outcome and respecting the constraints. This problem can be solved by integer programming by creating necessary fixed-arc variables to find out which arcs are present in the final solution and commodity-arc variables to find out which commodities are being routed by which arcs. We discuss more about this problem in chapter 5

The above generic problem can be used to represent some special cases of network design problems. [MW84] list the following network design problems that can be modeled and solved by integer programming:

- Shortest Path Problems and Minimum Spanning trees
- Traveling Salesman Problem
- Vehicle Routing Problem
- Facility Location Problem
- Traffic Equilibrium

In addition to the above problems, we study Network Design with Service Requirements in chapter 5 and propose algorithms to solve them efficiently.

2 Dense Subgraphs: Introduction

A graph is a mathematical structure containing vertices and edges that is often used to represent different real-life scenarios. Besides very traditional applications in transportation, mapping, and logistics, graphs may also be used to describe many social, biological, financial, and technological systems. In these cases, vertices represent individuals, cells, proteins, components, and edges represent some kind of interaction between the vertices. As a result, *Graph Theory* is one of the most extensively researched areas in computer science.

Graph networks that arise in real-life applications have edges which are either weighted or unweighted. While unweighted edges simply represent some connection between two vertices, weighted edges can be used to indicate the importance of a connection in the graph, or the time required for traveling on a given edge, or the probability of an edge to occur in the network. The edges could be further directed or undirected: the former model one-way relationships, like the “follow" network in Twitter, while the latter are used for two-way connections, for instance, Facebook friendships.

Identification of dense areas is a very interesting problem in social network analysis. Intuitively, dense areas in a graph can be considered to be a subset of highly-connected vertices that correspond to regions where there is more interaction among the vertices. For instance, consider a network describing the interactions between various Internet Service Providers, exchange points, customers, and other related parties: identifying dense subgraphs in this network allows us to detect critical

points of failure, which could further help in planning for contingencies to mitigate unplanned service outages. Similarly, for social networks, dense subgraphs identify areas of common interests and communities. Many other examples where finding dense subgraphs is a key problem are detailed in [Lee+10] and in [For10].

The first few chapters in this dissertation deals with the search of dense subgraphs in large graphs. The content is organized as follows. Chapter 2 discusses the definition of the problem and gives an overview of both the historical and more recent approaches to the *Densest Subgraph Extraction (DSE)* problem. Section 3 reviews the existing literature, presenting the main exact approaches for computing an optimal solution of the DSE problem, and an existing heuristic algorithm known as *Greedy Peeling*. Section 3.1 introduces a new algorithm called the *Hybrid* algorithm that is built on top of *Greedy Peeling* and an exact algorithm. All algorithms are computationally tested in Section 4 on a large set of graph instances taken from the literature including both unweighted and weighted graphs. Finally, Section 9.2 gives a summary and draws some conclusions.

We list three main contributions in this disseration pertaining to dense subgraph discovery. From a practical viewpoint, we introduce a simple heuristic algorithm that is built on top of the greedy heuristic and any exact method. Our proposed algorithm is typically very fast, produces solutions that improve over the greedy solution, and gives us near-optimal solutions.

From a theoretical point of view, we present a simple graph instance where the *Greedy Peeling* algorithm approaches its worst-case performance. To the best of our knowledge, there is only one other example in the literature showing a similar behavior of the *Greedy Peeling* algorithm (see [GT15]) but the example we present is simpler than the existing one. Besides, the example provided in [GT15] refers to a disconnected graph, and could be efficiently tackled by considering each connected component, one at a time. On the contrary, our example is a connected graph, for which we show that the *Greedy Peeling* algorithm achieves the

theoretical worst case performance.

Finally, from a computational perspective, we present a thorough experimental analysis, that is by far the most extensive in the literature for this class of problems. While most of the previous works in the literature have dealt with small or medium sized instances, in this dissertation we make a considerable step forward concerning the instance size by considering graphs with tens of millions of vertices and hundreds of millions of edges. Our computational study shows that the practical performance of the *Greedy Peeling* is much better than its theoretical guarantee, and that a further improvement can be achieved with limited computational effort.

2.1 Definition of the problem

In this section we give a formal definition of the problem. Let $G = (V, E)$ be an unweighted, undirected graph with vertex set V and edge set E. Throughout the text, we will assume that G is a simple graph, i.e., there are no multiple edges connecting the same pair of vertices. The *density* of G, sometimes referred to as *average degree*, is defined as

$$f(G) = \frac{|E|}{|V|}, \tag{2.1}$$

and corresponds to the ratio between the number of edges and the number of vertices in the graph.

For a given subset of vertices $S \subseteq V$, we define $E(S)$ as the induced set of edges, i.e., $E(S) = \{(u, v) \in E : u \in S, v \in S\}$, and $G(S) = (S, E(S))$ as the subgraph *induced* by S. When no confusion arises, we will write that set S has a density

$$f(S) = f(G(S)) = \frac{|E(S)|}{|S|} \tag{2.2}$$

Given an unweighted graph $G = (V, E)$, the *Densest Subgraph Extraction* (DSE) problem requires to determine a subset $S \subseteq V$ of vertices that

induces a subgraph of maximum density. Although it can be easily proved that there always exists an optimal solution to the DSE problem inducing a connected subgraph, we do not make any assumption on the input graph.

As already mentioned, in many applications each edge $(u, v) \in E$ has a positive *weight* w_{uv}, which could, for instance, be used to represent the importance of a relationship between two vertices in the network. Weighted graphs can also be used to model a unique scenario where the actual edge set is unknown and each potential edge has an associated non-negative probability. In this probabilistic setting, one is interested in finding a subgraph that has a large probability to be the one with maximum density. This leads to a natural extension of the density definition in (2.1) to the edge-weighted graphs as

$$f^w(G) = \frac{\sum_{(u,v)\in E} w_{uv}}{|V|}. \tag{2.3}$$

Similarly, we can define the weighted density for a given subset $S \subseteq V$ of vertices.

The aforementioned density definitions are valid for undirected graphs only. For directed graphs, different definitions are typically used and we refer the interested reader to [Cha00; KS09].

The DSE problem has been studied since the early 1980s. Though this problem is fundamentally an unconstrained non-linear optimization problem, it can still be solved efficiently. Indeed, a flow-based algorithm to get an optimal solution of the problem for unweighted graphs was introduced in [PQ82] and it requires utmost $|V|$ max-flow (min-cut) operations on a network of $|V| + 2$ vertices, i.e., it runs in polynomial time. Later, an alternative flow-based algorithm with better computational complexity was introduced in [Gol84]. This algorithm determines the densest subgraph in only $\mathcal{O}(\log(|V|))$ max-flow operations and can easily be extended to weighted graphs. Finally, a parametric max-flow algorithm which can solve the DSE with a single max-flow computation

was given in [GGT89]. This parametric max-flow algorithm improves upon the complexity of the previous method described in [Gol84] by a factor of $\log(|V|)$.

Though solvable in polynomial time, computing densest subgraphs using flow-based algorithms could be very time consuming for very large graphs. Thus, when real-world applications with millions of vertices and edges are considered, one has to resort to heuristics. One of the most important heuristic algorithms for the DSE problem is the *Greedy Peeling* introduced in [Asa+00]. Besides being very fast in practice, this algorithm has nice theoretical properties. It has been proved in [Cha00] that this algorithm has a worst-case 2-approximation, i.e., the density of the subgraph found by *Greedy Peeling* is at least half of the density of the optimal subgraph. The algorithm can be implemented to have time complexity of $\mathcal{O}(|E| + |V|)$ in case of unweighted graphs and $\mathcal{O}(|E| + |V| \log(|V|))$ in case of weighted graphs. Finally, we mention a variant of the *Greedy Peeling* algorithm, introduced in [BKV12], that can be implemented in a distributed way and for which the input is not stored, in order to reduce the memory requirement. This algorithm makes $\mathcal{O}(\log(|V)|)$ passes over the input graph and uses $\mathcal{O}(|V|)$ main memory, and has a worst-case approximation equal to $(2 + 2\epsilon)$ for any $\epsilon > 0$.

In some applications, additional constraints are imposed to limit (either from below or from above) the size of set S; in this case, the resulting problem becomes an $\mathcal{NP}$-hard problem. An extensive discussion on finding dense subgraphs with size bounds can be found in [AC09].

Many alternative definitions of density have been proposed in the literature. Indeed, the average-degree definition may produce subgraphs that have a large number of vertices, and are not extensively connected. For instance, a clique, which is intuitively a dense subgraph, might not be the densest subgraph according to the average degree, as another larger and loosely connected subgraph could produce a bigger ratio according to (2.1). Figure 2.1, shows an example in which the whole graph corresponds to the densest subgraph, with a density of $\frac{18}{7} = 2.57$, although

a clique exists (defined by the vertices in the dashed circle) that has a density equal to $\frac{15}{6} = 2.5$. Additional considerations about the downsides of using definition (2.1) as a metric to find the dense subgraphs are given in many papers from the literature. A different density metric, called *quasi-clique*, was introduced in [Tso+13]; according to this definition, the density of graph $G = (V, E)$ is given by $f(G) = |E(S)| - \alpha\binom{|S|}{2}$, where α is a tuning parameter. The authors in [Tso+13] claim that *quasi-clique* metric is better than *average-degree*, as it was shown that *quasi-clique* produces subgraphs that are tightly connected and smaller. In the same vein as [Tso+13], authors in [HS18] proposed another density metric called *discounted average degree* as $f(S) = \frac{|E(S)|}{|S|^\beta}$, where β is a parameter that can be chosen to affect the size of the desired subgraph. They also give four desirable properties of a density metric and show that their *discounted average degree* metric performs well on satisfying those four properties. Other than these two definitions, also depending on the type of graph, there have been many other proposed definitions of density, including edge ratio, triangle density, and triangle ratio, and others (see [ARS02; BBH11; CS12; Tso14]).

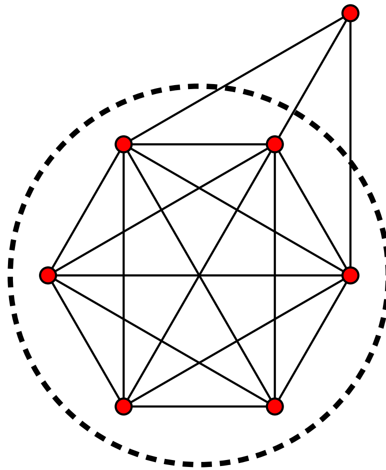

Figure 2.1: A small example to understand in which a clique is not the densest subgraph

Despite these alternatives, there is no clear consensus on using any of them as standard, and average degree remains the most common and accepted.

3 Algorithms to find dense subgraphs

In this section we discuss solution approaches for the DSE problem that have been proposed in the literature. The next subsection describes an exact algorithm and a mathematical formulation of the DSE, while Section 3.0.2 presents a greedy heuristic and analyzes its theoretical performance.

3.0.1 Exact algorithms

The first exact algorithm we consider is the *Goldberg's* algorithm which has been introduced in [Gol84] and is a relatively fast exact algorithm to compute the densest subgraph in a given graph G. For the sake of completeness, we report the algorithm's pseudo code in Figure 3.1. The algorithm iteratively guesses the solution value, solves a max-flow problem on an augmented network, and updates the value of the guess.

Figure 3.2 shows an illustration of an augmented network for a given guess g. The vertex set in the network is $V \cup \{s, t\}$, i.e., there are $|V| + 2$ vertices. Each edge in G is replaced by two reverse arcs with unit capacity. In addition, there is an arc from vertex s to each vertex $v \in V$ with capacity $|E|$, and an arc from each vertex $v \in V$ to vertex t with capacity $(|E| + 2g - d_v)$, where d_v is the degree of vertex v with respect to G.

At each iteration, the algorithm defines the augmented network $A(g)$ associated with the current guess g and computes a max $s - t$ flow (minimum cut) on this network. Depending on whether the minimum cut

```
Goldberg's: input = G(V, E)
initialize: ℓ := 0, u := |E|, S^E = ∅;
while u − ℓ ≥ 1/(|V| (|V|−1)) do
    g := (u+ℓ)/2;
'' define the augmented network A(g) associated with g;
    find the minimum cut (S, T) in A(g);
    if S = {s} then u := g
    else
        l := g;
        S^E := S \ {s};
end while
return S^E
```

Figure 3.1: *Goldberg's* algorithm

isolates vertex s, or instead separates the vertices in V in two nonempty subsets, the current g value reveals itself either a lower or an upper bound on the optimal density. The algorithm updates these bounds accordingly, until the difference between lower and upper bound is below some threshold.

It was proved in [Gol84] that, as the optimal g value can only take a finite set of values in the interval $[0, |E(S)|]$, the binary search converges to the optimal value and the number of iterations is bounded by $\mathcal{O}(\log(|V|))$. There are many efficient algorithms for solving max-flow problem (see, e.g., [Hen+18]). Using the Push-Relabel algorithm (see [GT88]), the max-flow problem can be solved in $\mathcal{O}(|V|^3)$ time, producing an overall $\mathcal{O}(\log(|V|)\,|V|^3)$ time complexity for *Goldberg's* algorithm.

A completely different exact solution method has been proposed in [Cha00]. This approach describes the DSE problem by means of a *Linear Programming* (LP) model, that can be solved using any general-purpose LP solver. The LP model can easily be extended to the weighted case with minor modifications. The model has $|V| + |E|$ variables and two

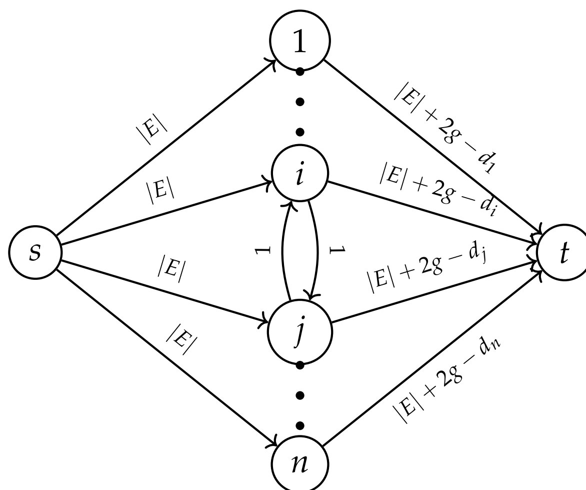

Figure 3.2: Augmented network $A(g)$, courtesy of [Gol84]

constraints per edge, i.e., its size is polynomial in the size of the input graph. Despite this, the constraint matrix of the formulation can be massive and the memory requirements to solve the model can be prohibitive for large graph instances. Typically this produces computational performances that are worse than those of the flow-based *Goldberg's* algorithm discussed above. However, the LP model provides a good foundation for finding dense subgraphs in directed graphs and its related proofs as discussed in [Cha00].

3.0.2 *Greedy Peeling* algorithm

For very large graphs, the application of the exact algorithms described in the previous section may require large memory and long computational times. This is where heuristic approaches can be used for getting reasonably good solutions quickly. The heuristic algorithm described in this section produces subgraphs whose density is usually close to an optimal one.

As the objective of DSE is to find a subgraph with best average degree,

the algorithm consists of starting with the initial graph and removing, one at a time, a vertex with the smallest degree in the current graph. The resulting algorithm, called *Greedy Peeling*, is described in Figure 3.3 and can be naively implemented to run in $\mathcal{O}(|V|^2)$ time. To prove the time complexity it is enough to observe that there are n iterations; each iteration requires $\mathcal{O}(|V|)$ time to find the vertex u with minimum degree with respect to the current subgraph (breaking ties arbitrarily), and another $\mathcal{O}(|V|)$ time to update the subgraph once u has been removed. A more efficient implementation can be obtained using a "degree-lists" data structure, in which a list is defined for each possible value of the degree of a vertex. All vertices with same degree are placed in the same list and lists are ordered by increasing degree. Using this data structure, the determination of the next vertex u to be removed can be done in constant time, taking an arbitrary vertex in the first non-empty list. Since removing vertex u decreases the degree of its neighbors by one unit, updating the graph (essentially data-lists) can be done by moving each neighbor of u from its current list to the previous one (i.e., to the list with degree one less than current degree). Since the number of vertex movements among the lists is equal to the number of edges of G, the time complexity of the algorithm is $\mathcal{O}(|E| + |V|)$. The results in this paper (see Section 4) correspond to this implementation of the algorithm.

Greedy Peeling: input = $G(V, E)$

initialize: $n := |V|$, $S_n := V$;
for $i = n$ **to** 1 **do**
    let u be the smallest degree vertex in $G(S_i)$;
    $S_{i-1} := S_i \setminus \{u\}$;
endfor
$S^H \in \arg\max_{i=1,\ldots,n} f(S_i)$;
return S^H

Figure 3.3: *Greedy Peeling* algorithm for the unweighted case.

Extension to the weighted case.

The *Greedy Peeling* algorithm can easily be extended to the weighted case by selecting, at each iteration, vertex u as the one having the minimum *weighted-degree*, i.e., the weighted sum of all the incident edges with respect to the current subgraph. However, the linear time complexity of the algorithm is not preserved because the degree-lists data structure cannot be used for graphs with general weights. Using Fibonacci heaps to determine, at each iteration, the minimum weighted-degree vertex, the algorithm runs in $\mathcal{O}\big(|E| + |V|\log(|V|)\big)$, see [Cha00]. The degree-lists implementation could also be used to determine weighted dense subgraphs, similar to the unweighted case, if weights are either integer numbers or are all scaled to integers. Assume the weights are integers and let u be the vertex that has currently been selected for removal. The weighted-degree of each neighbor of u, say vertex v, is decreased by an amount w_{uv}, instead of one as in the unweighted case. However, using degree lists may yield to a considerable worsening in the performance of the algorithm as the number of lists to be considered is bounded by the maximum weight degree of all vertices, i.e., it is pseudo-polynomial in the size of the input (and is strongly dependent on the number of significant digits in the weight values, if they are not integers).

To avoid this issue, we use binary heaps to implement the *Greedy Peeling* algorithm for the weighted case. A binary heap data structure is a complete binary tree which satisfies heap ordering. In particular, we use the min-heap property, which requires that the value of each node in the tree is greater than or equal to the value of its parent node. Initially, we compute the weighted-degree of each vertex and insert all the vertices in a heap data structure satisfying the min-heap property. At each iteration, determining the next vertex to be removed can be done in constant time, as the minimum value is associated with the root node of the tree. Once a vertex has been removed, updating the weighted degree of its neighbors and rearranging those vertices in the heap can be done in $\mathcal{O}(\log(|V|))$ time. In the following, we will report results for

this implementation, which works for both rational and integer weights, and is very fast in practice even for large graphs.

Worst-case analysis

The theoretical performance of the *Greedy Peeling* algorithm was analyzed in [Cha00] (and in [Asa+00] for a constrained version of the DSE problem), where the worst-case performance ratio of the algorithm was proved to be equal to 2. To the best of our knowledge, the only example for which the approximation is asymptotically tight has been given in [GT15].

In the following we present a simpler instance, where the worst-case performance ratio is approached. In addition, while the example reported in [GT15] is based on a disconnected graph, the following instance refers to a connected graph. To the best of our knowledge, this is the first example showing that the worst-case performance ratio of the algorithm may be hit for connected graphs as well. This result provides a relevant piece of information about the performance of the *Greedy Peeling* algorithm; indeed, it shows that, given a disconnected graph, the worst-case approximation provided by the algorithm cannot be improved by sequentially considering all the connected components, one at a time.

The instance shown in Figure 3.4 is a graph G which has two vertices u and v connected by an edge; both vertices u and v are also connected to additional $2k$ vertices indexed by {1, . . . , 2k} by $4k$ edges. Vertices u and v are also connected by a path consisting of another set of t vertices and $t+1$ edges. Thus, graph G has $2+2k+t$ vertices and $1+4k+t+1=4k+t+2$ edges.

At the first iteration the *Greedy Peeling* considers the full graph, which has a density equal to $f(G)=\frac{4k+t+2}{2k+t+2}$. In the first $2k$ iterations, all vertices but u and v have degree 2. Breaking ties by lowest index, the algorithm removes, in turn, vertices $1, 2, \ldots, 2k$. Each vertex removal induces the elimination of two edges from the remaining subgraph; it is easy to see that the resulting density cannot be larger than $f(G)$. When vertices 1, 2, . . . , $2k$ have been removed, the remaining subgraph is a cycle spanning

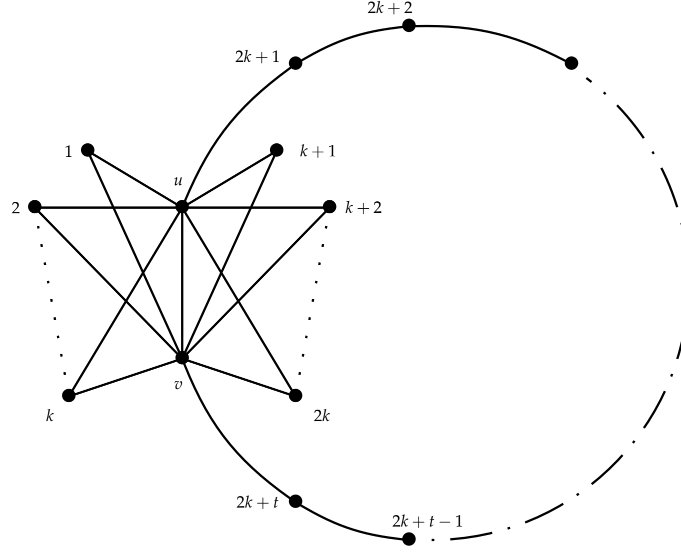

Figure 3.4: Bad connected instance for the *Greedy Peeling*. The graph has $2k + t + 2p$ vertices and $4k + t + 2$ edges.

$t + 2$ vertices. Regardless the order in which the vertices are removed, the algorithm encounters subgraphs having a density smaller than the initial one. Thus, the *Greedy Peeling* returns a heuristic solution with value $f^G = \frac{4k+t+2}{2k+t+2}$.

An optimal solution is defined by vertex set $\{1, 2, \ldots, 2k\} \cup \{u, v\}$. The induced subgraph has $2k + 2$ vertices and $4k + 1$ edges, hence the optimal solution value is $f^* = \frac{4k+1}{2k+2}$. Thus, the ratio between the optimal and the approximate solution values is given by

$$\frac{f^*}{f^G} = \frac{\frac{4k+1}{2k+2}}{\frac{4k+t+2}{2k+t+2}} = \frac{(4k+1)(2k+t+2)}{(2k+2)(4k+t+2)} \tag{3.1}$$

Taking $t = k^2$ we have that $\frac{f^*}{f^G}$ is arbitrarily close to 2 for sufficiently large values of k.

Finally, observe that a simple adaptation to the weighted case of the worst-case analysis given in [Cha00] shows that, also in this case, the

Greedy Peeling returns a solution value which is at least half of the optimal density. As graph G is a weighted instance with all weights equal to 1, this results shows that the worst-case approximation ratio is tight in the weighted case as well.

3.1 *Hybrid* algorithm

***Hybrid*:** input = $G(V, E)$

// *Greedy Peeling*

$S^1 :=$ ***Greedy Peeling***$\big(G(V, E)\big)$;

// *Expansion* phase

$S^2 := \{v \in V : (u, v) \in E \text{ for some } v \in S^1\}$;

$E^2 := \{(u, v) \in E : \ u \in S^2, v \in S^2\}$;

// *Exact* phase

$S^H :=$ ***Exact***$\big(G(S^2, E^2)\big)$;

return S^H

Figure 3.5: *Hybrid* algorithm.

In this section we present a *Hybrid* algorithm that combines the *Greedy Peeling* and an exact algorithm to improve the greedy solution value. The algorithm is given in Figure 3.5 and consists of three phases, namely *Greedy Peeling*, *Expansion* phase, and *Exact* phase. The first phase corresponds to the execution of the *Greedy Peeling* algorithm discussed in Section 3.0.2 and is intended to quickly produce an initial solution. Using this initial greedy solution, the *Expansion* phase obtains a “core” subgraph, which is likely to contain either all or most of the vertices in an optimal solution. Finally, the *Exact* phase solves the DSE problem on

the core using an exact algorithm, for instance, the flow-based *Goldberg's* algorithm or the LP approach described in Section 3.0.1.

The *Expansion* phase takes in input a subset of vertices S^1, possibly identified by the *Greedy Peeling*, expands the vertex set by adding all those vertices that are neighbors of one vertex in S^1, and defines the induced edge set E^2. An implementation of this phase is described in Figure 3.6. Set S^2 includes all the vertices that are currently included in the expanded graph. Before the *Expansion* phase, $S^2 = \emptyset$. In the *Expansion* phase we considers all vertices in S^1, one at a time. For each $u \in S^1$, we consider all its neighbors; if the current neighbor v is in $S^1 \cap S^2$, we add the edge (u, v) to E^2. If $v \notin S^2$, we add vertex v to S^2 and edge (u, v) to E^2, and scan all neighbors of v; for each neighbor k

```
Expansion: input = S^1, V, E
S^2 := ∅, E^2 := ∅;
//consider each vertex u in the input solution
for each u ∈ S^1 do
    S^2 := S^2 ∪ {u};
    // add all neighbors of u
    for each v ∈ V : (u, v) ∈ E do
        if v ∈ S^1 then
            if v ∈ S^2 then E^2 := E^2 ∪ {(u, v)};
        else
            S^2 := S^2 ∪ {v}, E^2 := E^2 ∪ {(u, v)};
            // add edges between vertices that both are in S^2 \ S^1
            for each k ∈ S^2 \ S^1 : (v, k) ∈ E do E^2 := E^2 ∪ {(v, k)};
        endif
    endfor
endfor
return G(S^2, E^2)
```

Figure 3.6: *Expansion* phase.

that is currently in set S^2 we also add an edge (v,k) to E^2.

Figure 3.7 gives an example of the *Expansion* phase. The original graph has 12 vertices and $S^1 = \{5,6,7,8\}$. At first, $S^2 = \emptyset$. We can start at vertex $u = 5$ which makes $S^2 = \{5\}$ and consider the first of its neighbors, i.e., vertex $v = 2$. From the algorithm, we can add vertex 2 to S^2 and the edge $(2,5)$ to E^2. Now, we scan the neighbors of 2 and we can add an edge to E^2 if any of the neighboring vertices of 2 are present in S^2. Since no new neighboring vertices of 2 (essential vertex 1) are present in S^2, we do not add any new edges to E^2. So we have $S^2 = \{5,2\}$ and $E^2 = \{(2,5)\}$. Then, we examine the other neighboring vertices of 5 namely $6,7$ and 8. As all these vertices belong to S^1 and none of them are in S^2, no action is taken. Then, we move on the next member in S^1, i.e. $u = 6$. We add 6 to S^2 and examine the neighbors of 6. We have vertex $v = 3$ that can be added to S^2 and the edge $(3,6)$ can be added to E^2. Now, $S^2 = \{5,2,6,3\}$ and $S^2 \setminus S^1 = \{2,3\}$, implying that edge $(3,2)$ has to be added to E^2. Since no other edge can be added, we then move on to the next neighbor of 6, namely 5. When considering this vertex, edge $(6,5)$ can be added to E^2 as 5 is in both S^1 and S^2. As all the neighbors of 6 have been considered, we move onto the next vertex in S^1, i.e. 7. We continue doing the above process for all the members in S^1 until we get the expanded subgraph, shown in Figure 3.7(b)

In the third phase, an exact algorithm is applied to the graph obtained by the *Expansion* phase. Typically this graph is much smaller than the original one, allowing a fast execution of the exact algorithm. In addition, if the flow-based algorithm is used, the greedy solution value, combined with the 2-approximation guarantee of the method, produces good initial lower and upper bounds for the value of the density, which can be used to speed up the binary search. The biggest caveat is that there are instances for which the *Greedy Peeling* produces very large subgraphs. In this situation, the *Expansion* phase may require a very long computing time, and often returns the original graph, making this approach impractical.

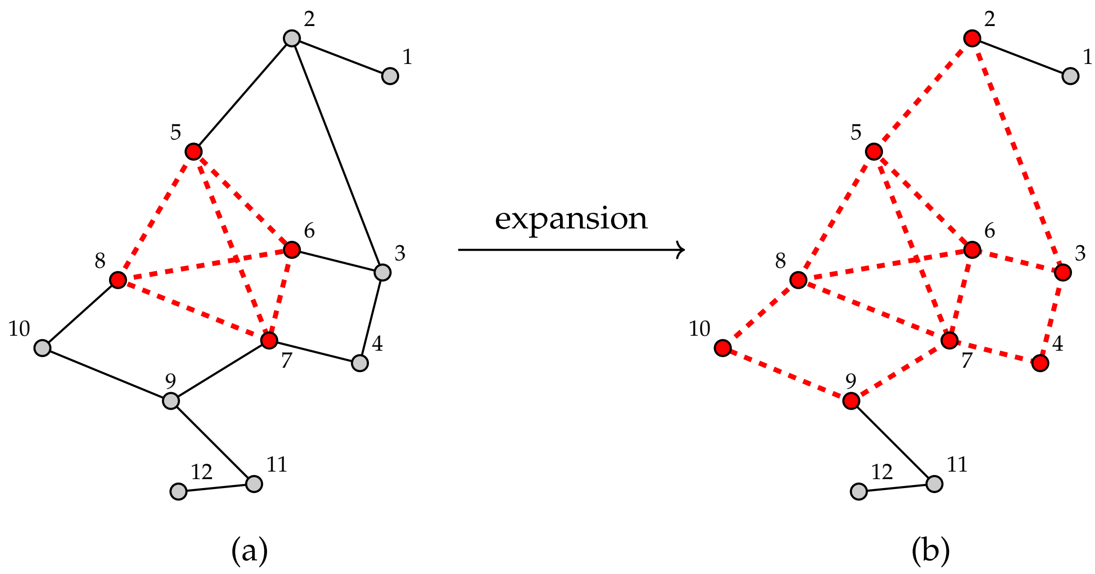

Figure 3.7: *Expansion* phase example

4 Dense Subgraphs: Computational experiments

4.1 Setup and programming

All the algorithms described in this paper were implemented in C++ using standard containers, like `std::vector`, `std::queue`. We used the GCC compiler with a high level of optimization enabled (`-O3`). All our experiments were executed on a computer equipped with an Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10GHz CPU and 16 GB of RAM; all computing times (t) given below are expressed in milliseconds.

In the following we report the results obtained using the three algorithms, namely:

- The *Greedy Peeling* discussed in Section 3.0.2.
- The *Hybrid* algorithm of Section 3.1.
- The flow-based exact algorithm (*Goldberg's* algorithm) of Section 3.0.1. This algorithm embeds a push-relabel algorithm to compute the max-flow (min-cut) with $\mathcal{O}(|V|^3)$ time complexity. It should be noted that this algorithm requires to construct an augmented network which has more than twice the number of edges than the original graph network. As a result, the augmented network could occupy very large space in memory, and hence the algorithm may fail for memory requirement on very large instances.

We analyzed both weighted and unweighted instances (see below). For weighted instances, the *Greedy Peeling* was implemented using binary

heaps as this solution turned out to be much more efficient than using the degree-list implementation. As to the *Goldberg's* algorithm, it required very minor modifications for handling the weighted case as well. The *Hybrid* algorithm uses the *Greedy Peeling* and the *Goldberg's* algorithm as modular components to create and solve the expanded subgraph respectively, while the *Expansion* phase clearly is not affected by the presence of weights on the edges.

4.2 Testbed

All the instances, both unweighted and weighted, were taken from Suite Sparse Collection [DH11]. To select a meaningful set of instances, we considered graphs that:

(i) are classified as *undirected graph* or *undirected weighted graph* or *undirected graph with communities* or *undirected random graph*;

(ii) have at least 20,000 vertices;

(iii) have at most 65,000,000 vertices and 150,000,000 edges; and

(iv) only have positive weights (for weighted instances).

As informed in Section 2.1, the definition of density for directed graphs is significantly different than for undirected graphs and hence (i) we only considered the latter. Since the DSE is solvable in polynomial time, and hence optimal solutions are obtained with limited effort for small graphs, (ii) we chose to only consider instances that are "not too small" and may therefore be challenging for our algorithms. We also imposed some upper-bounds on the size of the graphs as any graphs which are bigger than the ones mentioned in (iii) can not be solved by any algorithm on our machine as all of them run out of memory. And finally, (iv) we only considered graphs with positive weights, as all algorithms discussed in this paper do not have a straightforward extension when negative

weights are considered. For instance, *Goldberg's* algorithm would fail in case of negative weights.

This produced a testbed with 170 instances. The benchmark includes 50 census-based weighted graphs (like xx2010 in Table 4.8) that have very similar characteristics. To avoid presenting very similar results, we decided to consider only the ten largest among these instances. In addition, we have also considered three large directed graphs (called Wikipedia instances), that were present in the computational analysis in [Tso+13]; for these instances, minor modifications were required, e.g., converting directed arcs to undirected edges and removing duplicated edges. Finally, we do not present the results on some graphs where the greedy algorithm fails.

The majority of the graphs in our testbed are unweighted and hence we have further partitioned them into different buckets, depending on their size. The *Medium* bucket contains those instances which have less than 1,000,000 vertices. The *Large* bucket contains instances having more than 1,000,000 vertices but less than 10 million vertices and less than 50,000,000 edges. Finally, the *Massive* bucket includes all the remaining instances.

4.3 Analysis on instances in the *Medium* bucket

In this section we report the outcome of our computational experiments on the instances in the first bucket, that contains 41 instances.

Table 4.1 gives the results and reports, for each instance, the following information:

- The name of the instance and the main characteristics of the graph.
- For the *Greedy Peeling* algorithm: the required computing time t_G and the associated density value f_G.
- For the *Hybrid* algorithm: the computing time for the *Expansion* phase and for the *Exact* phase (t_2 and t_3, respectively), the overall

computing time t_H of the algorithm and the density value f_H of the best solution found.

- For the exact algorithm (*Goldberg's* algorithm): the required computing time t_E and the density value f^*.

If an algorithm runs out of memory during its execution, we report the failure by '–'. The bold numbers in the table indicate the best density found. In case of ties, the density of the fastest algorithm is boldfaced.

Results in Table 4.1 show that *Goldberg's* algorithm can handle this set of instances quite efficiently: the required computing time is equal to 162 seconds on average and no failure was experienced due to memory reasons. The *Greedy Peeling* algorithm, though having a worst-case performance ratio equal to 2, gives a very tight approximation on the optimal density in practice, as the average gap with respect to the optimal density is 3.12%. In addition, this algorithm is very fast, the average CPU time being around 0.09 seconds.

The *Hybrid* algorithm has good performances, as it improves over the greedy solution in 27 cases, but it runs out of memory for instance mycielskian17; on the remaining 40 instances, the average percentage gap of the algorithm is about 1.14%. The table shows that there are a number of instances for which the *Hybrid* algorithm performs poorly in terms of computing time. In Table 4.2 we report all the instances where the ratio of $\frac{t_H}{t_E} > 0.75$ and where the *Hybrid* algorithm runs out of memory. For each such instance, the table gives:

- The name of the instance.

- The number of vertices $|S^1|$ in the subgraph produced by *Greedy Peeling* and the ratio between $|S^1|$ and the total number of vertices V.

- The number of vertices and edges ($|S^2|$ and $|E^2|$, respectively) in the expanded subgraph and the ratio between $|S^2|$ and the total number of vertices V.

Table 4.2 shows that the *Hybrid* algorithm encounters difficulties while dealing with instances where the solution produced by *Greedy Peeling* has almost the same number of vertices as the whole graph. And sometimes, even when *Greedy Peeling* produces a smaller and more compact solution, the *Expansion* phase produces either the original graph or almost the original graph. For these specific instances, which are identified by the ratio $|S^2|/|V|$ being close to 1, the *Expansion* phase may be time consuming, and the application of *Goldberg's* algorithm after *Expansion* requires similar computing time as applying *Goldberg's* algorithm to the original instances. Thus, the *Hybrid* algorithm may overall be even slower than the direct application of *Goldberg's* algorithm on the initial graph instances. The average computing time taken by the *Hybrid* algorithm for instances in the *Medium* bucket is around 115 seconds; if we exclude the 14 pathological instances listed in Table 4.2, time take by the *Hybrid* algorithm falls to around 21 seconds.

4.4 Tuning of the algorithm

In this section we present some additional results for evaluating variants of the *Hybrid* algorithm. In particular, we consider:

- `H1`: This algorithm is aimed at evaluating the effect of the *Expansion* phase. In this scheme we simply disabled the *Expansion* phase of the *Hybrid* algorithm and executed *Goldberg's* algorithm on the subgraph produced by *Greedy Peeling*. However, since the output of the latter consists of a set of vertices only (S^1), the associated edges have to be reconstructed and stored.
- `H2`: This algorithm is used to evaluate possible solution improvements obtained by repeatedly performing the *Expansion* and *Exact* phases. The algorithm operates in two steps: first, it executes the *Hybrid* algorithm and stores the associated solution. Then, this solution is expanded again using the *Expansion* phase and *Goldberg's* algorithm is invoked on the resulting subgraph.

- H3: This algorithm is intended to evaluate if a different way to expand could produce a larger graph, allowing *Goldberg's* algorithm to determine a subgraph with better density. In particular, given the set S^1 of vertices produced by *Greedy Peeling*, this algorithm executes the *Expansion* phase twice in sequence. In this way, the graph which is used as input for *Goldberg's* algorithm includes the neighbors of the vertices in S^1 and also the neighbors of the neighbors.

In these experiments, we consider again the 40 instances in the *Medium* bucket but mycielskian17. Table 4.3 gives results for the three schemes above, as well as for the *Greedy Peeling* and for the *Hybrid* algorithm described in Section 3.1. For each algorithm, we report the average values of the computing time (in milliseconds) and average percentage gap with respect to the optimal solution. The statistics are computed with respect to all instances in the *Medium* bucket but the instance mycielskian17.

These results show that variants H2 and H3, while requiring additional computational effort when compared to the *Hybrid* algorithm, only produce negligible improvements in the solution quality.

The situation is less clear for H1, which is computationally less expensive than the *Hybrid* algorithm, but also finds solutions of lower quality; for this reason, we analyzed the performance of these two algorithms on a restricted subset of instances. According to the results in Table 4.2, the *Hybrid* algorithm performs badly if *Goldberg's* algorithm is applied to a graph whose size is comparable with the original one. For this reason, we removed all instances for which our solution approach has small probability of being successful, and selected the instances for which $|S^2|/|V| < 0.85$ (resp. $|S^1|/|V| < 0.85$ for H1). Table 4.4 reports the statistics with respect to these instances only. As the number of these instances depends on the algorithm, we also report the number of instances that are used for comparison. On this restricted benchmark, H1 has an average percentage gap of around 3% and an average computing time of 18 seconds, while the *Hybrid* algorithm has a slightly larger average computing time (around 21 seconds), but the average percentage gap is

almost halved (around 1.7%), thus showing the robustness of our design choices.

4.4.1 Disconnected graphs.

As previously mentioned, when the DSE problem is solved for a disconnected graph, there always exists an optimal solution corresponding to a connected subgraph. Hence, instead of running an algorithm on the whole graph, a possible strategy involves executing it on each component separately. Although checking for connectivity in a given graph is an easy task, its practical difficulty depends on the size of the graph and on the way it is described. For example, when the graph is described as a list of edges, extracting and storing the connected components requires an increased memory usage, which can be a problem for huge graphs.

Analyzing instances in our benchmark, we found that most of them are connected graphs. As to the remaining disconnected ones, each of them typically has few large components with many vertices, and a very large number of small components that have very few vertices. Hence, in order to evaluate possible improvements obtained by considering each component separately, we tested both *Greedy Peeling* and *Hybrid* algorithm where we: i) first detect the connected components of the input graph by using *Depth-first search* algorithm, and then ii) run the specific algorithm on the largest connected component only.

Table 4.5 reports the statistics for the 12 instances in the *Medium* bucket that are disconnected. It shows us the average computing time (in milliseconds) and the average percentage gap for both *Greedy Peeling* and *Hybrid* algorithm executed on the original graph and on the biggest component in the said original graph. The table shows that component detection has limited impact in terms of computing time, as for both algorithms we observe a small reduction (a few milliseconds) of the average time. However, we also observe a non-negligible worsening of the solution quality (the average gap increases by around 4% in both cases). This is due to the fact that for one instance, the optimal

subgraph is present in a smaller component and not in the biggest one. This analysis shows us that, in order to find the densest subgraph in a disconnected graph, we can not restrict ourselves to applying the algorithms on the largest component of the graph. This leads to an increase in the time taken to find the densest subgraph as we have to run the algorithms on all the large components of the disconnected graph.

4.5 Results on instances in the *Large* and *Massive* buckets

In this section we present the results of our experiments on instances in the *Large* and *Massive* buckets. Based on the outcome of the results in the previous sections, we do not run the *Hybrid* algorithm for those instances where the greedy solution (or the expanded subgraph) is almost as large as the original graph. In particular, we removed the graphs for which $\frac{|S^2|}{|V|} > 0.85$, namely the instances in the series delaunay, hugebubbles, hugetrace, and hugetric, as well as instances 333SP, adaptive, AS365, channel-500x100x100-b050, M6, NACA0015, and NLR. Note that $|S^2|$ can be computed in negligible time before performing the *Expansion* phase, simply scanning all the edges that are incident to vertices in the greedy solution.

Table 4.6 addresses the instances in the *Large* bucket and shows that, similar to instances in the *Medium* bucket, *Hybrid* algorithm consistently improves upon the density value produced by *Greedy Peeling*, frequently producing an optimal solution. The *Hybrid* algorithm was able to find the optimal solution in 13 cases out of 21 instances, and in 12 out of these 13 cases it was faster than the *Goldberg's* algorithm. As for the 8 instances that are not solved to optimality, the associated average gap is around 3.5%. The average gap over all the 21 instances is around 1.3%, much smaller than that of the *Greedy Peeling*, which is around 6.7%. As for the computing time, *Greedy Peeling* just takes around 1.3 seconds on average, while the *Hybrid* algorithm takes 215 seconds on average. The *Goldberg's*

algorithm takes more than 1050 seconds on average for solving these instances to optimality.

In Table 4.7, we present the results of the three algorithms for the instances in the *Massive* bucket in our benchmark. These graph instances were derived from real-life applications like gene networks (kmer series), road networks, social networks, and others. It can be immediately seen that *Goldberg's* algorithm fails for all the instances due to memory limitation. For these instances, *Greedy Peeling* finds a dense subgraph within 10 seconds on average, despite running on some graphs having tens of millions of vertices and hundreds of millions of edges. The *Hybrid* algorithm consistently improves upon the greedy solution for most instances, the only exceptions being hollywood-2009, where both algorithms give the same solution, and soc-orkut, for which the *Hybrid* algorithm runs out of memory. Ignoring this last instance, the average computing time taken by the *Hybrid* algorithm is around 41 seconds, and the average improvement produced by this algorithm over the *Greedy Peeling* is around 10%.

4.6 Results on weighted instances

Finally, in this section we address the weighted instances and report the associated results in Table 4.8.

The *Greedy Peeling* performs very well, and finds a provable optimal solution in 9 out of 16 cases; for instance mawi_201512020000 it produces the same density value as the *Hybrid* algorithm, but optimality of the solution cannot be confirmed as the *Goldberg's* algorithm fails. The *Hybrid* algorithm improves over the greedy solution in 5 of the 6 remaining instances, in 4 of these cases finding a provable optimal solution. On average, the *Greedy Peeling* takes 2 seconds, while the *Hybrid* algorithm takes 28 seconds. On the other hand, *Goldberg's* algorithm fails to solve 2 instances and, for the remaining instances, it requires on average almost 98 seconds to find the optimal solution. By removing the two mawi

instances, we see that the average percentage gap of the *Greedy Peeling* is around 1.72%, which is reduced to less than 0.02% by the *Hybrid* algorithm.

Graph Properties			*Greedy Peeling*		*Hybrid*				*Goldberg's*	
Instance	$\lvert V\rvert$	$\lvert E\rvert$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
144	144,649	1,074,393	53	7.4416	30,114	259,526	289,694	7.4559	280,444	**7.4559**
598a	110,971	741,934	37	6.8043	5,066	35,843	40,947	**6.8792**	73,151	6.8792
as-22july06	22,963	48,436	3	**19.9423**	11	737	751	19.9423	1,317	19.9423
auto	448,695	3,314,611	181	7.4495	89,415	310,410	400,007	7.5211	622,512	**7.5213**
ca-CondMat	23,133	93,439	4	12.5000	<1	8	13	**13.3667**	2,336	13.3667
caidaRouterLevel	192,244	609,066	50	25.5167	9	223	282	**25.7750**	23,785	25.7750
citationCiteseer	268,495	1,156,647	96	12.0019	205	6,268	6,570	**12.1808**	59,115	12.1808
coAuthorsCiteseer	227,320	814,134	60	**43.0000**	4	58	123	43.0000	25,229	43.0000
coAuthorsDBLP	299,067	977,676	78	57.0000	5	74	158	**57.0690**	35,584	57.0690
com-Amazon	334,863	925,872	104	3.8327	2,163	8,674	10,942	**4.8041**	53,902	4.8041
com-DBLP	317,080	1,049,866	94	56.5000	6	77	178	**56.5652**	38,550	56.5652
coPapersCiteseer	434,102	16,036,720	296	**422.0000**	229	3,316	3,842	422.0000	205,527	422.0000
coPapersDBLP	540,486	15,245,729	358	**168.0000**	67	2,324	2,752	168.0000	233,183	168.0000
cs4	22,499	43,858	2	1.9493	247	8,059	8,309	**1.9526**	9,008	1.9526
dblp-2010	326,186	807,700	62	**37.0000**	4	15	82	37.0000	26,000	37.0000
delaunay_n15	32,768	98,274	4	**2.9991**	710	12,997	13,712	2.9991	14,375	2.9991
delaunay_n16	65,536	196,575	10	**2.9995**	2,835	50,002	52,848	2.9995	49,406	2.9995
delaunay_n17	131,072	393,176	23	**2.9997**	11,103	147,668	158,794	2.9997	145,617	2.9997
delaunay_n18	262,144	786,396	47	**2.9999**	44,140	394,457	438,645	2.9999	427,411	2.9999
delaunay_n19	524,288	1,572,823	96	**2.9999**	176,182	1,740,164	1,916,442	2.9999	1,768,799	2.9999
dictionary28	52,652	89,038	5	**12.5000**	1	4	11	12.5000	2,634	12.5000
fe_body	45,087	163,734	7	3.9043	2	159	168	3.9213	5,421	**4.0490**
fe_ocean	143,437	409,593	22	2.8734	6,533	49,005	55,561	2.8964	80,359	**2.8966**
fe_rotor	99,617	662,431	27	6.6571	12,459	146,689	159,176	**6.6920**	159,632	6.6920
fe_tooth	78,136	452,591	20	5.9171	2,319	25,546	27,885	5.9778	58,032	**5.9801**
loc-Brightkite	58,228	214,078	11	40.5571	12	492	515	**40.5591**	6,124	40.5591
loc-Gowalla	196,591	950,327	62	43.8000	174	11,902	12,139	**43.8018**	32,753	43.8018
luxembourg_osm	114,599	119,666	10	1.1548	2	2	15	1.2667	4,338	**1.5238**
m14b	214,765	1,679,018	79	7.8266	71,078	185,721	256,879	7.8694	238,330	**7.8694**
mycielskian15	24,575	5,555,555	101	**333.5567**	30,001	97,961	128,064	333.5567	107,600	333.5567
mycielskian16	49,151	16,691,240	322	**530.8705**	175,641	305,244	481,208	530.8705	344,396	530.8705
mycielskian17	98,303	50,122,871	1,092	**845.8977**	–	–	–	–	1,165,647	845.8977
rgg_n_2_15_s0	32,768	160,240	6	7.5500	<1	1	8	7.6522	3,336	**7.8947**
rgg_n_2_16_s0	65,536	342,127	16	7.6471	<1	2	19	**9.000**	7,824	9.0000
rgg_n_2_17_s0	131,072	728,753	39	8.0000	1	1	42	8.2083	20,552	**8.9200**
rgg_n_2_18_s0	262,144	1,547,283	87	10.0769	3	2	93	**10.4242**	45,015	10.4242
rgg_n_2_19_s0	524,288	3,269,766	190	8.9474	5	1	197	**10.1667**	125,960	10.1667
t60k	60,005	89,440	5	1.4905	1,036	91,590	92,632	1.4914	83,854	**1.4914**
usroads	129,164	165,435	19	1.5789	1	<1	21	1.6250	11,992	**1.7528**
usroads-48	126,146	161,950	18	1.5714	2	1	22	1.6250	14,238	**1.7528**
wing	62,032	121,544	9	1.9596	1,897	46,318	48,225	**1.9627**	53,894	1.9627

Table 4.1: Results on instances in the *Medium* bucket. All times are in milliseconds.

Graph Properties	*Greedy Peeling*		*Hybrid*		
Instance	$\lvert S^1\rvert$	$\lvert S^1\rvert/\lvert V\rvert$	$\lvert S^2\rvert$	$\lvert E^2\rvert$	$\lvert S^2\rvert/\lvert V\rvert$
144	137,542	0.9509	138,830	1,032,694	0.9598
cs4	22,498	1.0000	22,499	43,858	1.0000
delaunay_n15	32,767	1.0000	32,768	98,274	1.0000
delaunay_n16	65,535	1.0000	65,536	196,575	1.0000
delaunay_n17	131,071	1.0000	131,072	393,176	1.0000
delaunay_n18	262,143	1.0000	262,144	786,396	1.0000
delaunay_n19	524,287	1.0000	524,288	1,572,823	1.0000
fe_rotor	98,214	0.9859	98,971	658,472	0.9935
m14b	206,912	0.9634	210,693	1,647,651	0.9810
mycielskian15	9,078	0.3694	24,575	5,555,555	1.0000
mycielskian16	16,436	0.3344	49,151	16,691,240	1.0000
mycielskian17	28,496	0.2899	98,303	50,122,871	1.0000
t60k	59,866	0.9977	59,935	89,313	0.9988
wing	61,852	0.9971	61,994	121,461	0.9994

Table 4.2: Instances for which the *Hybrid* algorithm can take a very long time.

Instances	*Greedy Peeling*		`H1`		*Hybrid*		`H2`		`H3`	
	t	%gap	t	%gap	t	%gap	t	%gap	t	%gap
ALL	68	3.2015	87,320	2.1986	115,199	1.1427	205,276	1.1420	134,274	1.1419

Table 4.3: Average computing time and percentage gap for different variants of the *Hybrid* algorithm. All times are in milliseconds.

Instances	`H1`			*Hybrid*		
	# inst.	t	%gap	# inst.	t	%gap
SELECTED	29	18,531	3.0322	27	20,864	1.6928

Table 4.4: Average computing time and percentage gap for `H1` and *Hybrid* algorithm on a selected subset of instances. All times are in milliseconds.

	Original Graph				Biggest Component			
Instances	*Greedy Peeling*		*Hybrid* algorithm		*Greedy Peeling*		*Hybrid* algorithm	
	t	% gap	t	% gap	t	% gap	t	% gap
DISCONNECTED	41	5.5019	121	1.7912	34	9.8064	113	5.5338

Table 4.5: Performance of *Greedy Peeling* and *Hybrid* algorithm on disconnected graphs. All times are in milliseconds.

Graph Properties			*Greedy Peeling*		*Hybrid*				*Goldberg's*	
Instance	$\|V\|$	$\|E\|$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
as-Skitter	1,696,415	11,095,298	822	89.1810	2,303	37,273	40,399	**89.4009**	388,513	89.4009
asia_osm	11,950,757	12,711,603	1,342	1.7778	135	<1	1,478	1.7778	703,145	**1.8513**
belgium_osm	1,441,295	1,549,970	185	1.6000	15	<1	200	1.6000	77,872	**1.6750**
com-LiveJournal	3,997,962	34,681,189	3,129	190.9845	82	695	3,907	**193.5136**	1,226,155	193.5136
com-Youtube	1,134,890	2,987,624	341	45.5778	1,608	60,642	62,592	**45.5988**	157,970	45.5988
germany_osm	11,548,845	12,369,181	1,734	1.6250	133	<1	1,868	1.6667	784,833	**1.7500**
great-britain_osm	7,733,822	8,156,517	1,039	1.8710	93	1	1,134	**1.9583**	465,254	1.9583
italy_osm	6,686,493	7,013,978	743	1.6250	80	<1	824	1.6667	365,157	**1.7778**
netherlands_osm	2,216,688	2,441,238	298	1.6667	29	<1	328	**1.7143**	190,545	1.7143
packing-500x100x100-b050	2,145,852	17,488,243	640	8.5361	147,977	612,576	761,195	8.7361	2,931,714	**8.8078**
rgg_n_2_20_s0	1,048,576	6,891,620	415	11.1212	14	4	433	11.6250	276,226	**11.6346**
rgg_n_2_21_s0	2,097,152	14,487,995	930	9.3934	22	7	960	**11.9048**	667,290	11.9048
rgg_n_2_22_s0	4,194,304	30,359,198	1,937	10.5503	58	25	2,021	**12.550**	1,806,177	12.5500
road_central	14,081,816	16,933,413	3,285	1.6002	179	20	3,485	1.7750	6,231,763	**1.9029**
roadNet-CA	1,971,281	2,766,607	315	1.6743	48	233	597	**1.9677**	313,535	1.9677
roadNet-PA	1,090,920	1,541,898	177	1.6441	14	14	205	1.8571	234,657	**1.8783**
roadNet-TX	1,393,383	1,921,660	216	1.7656	17	7	241	**2.0769**	82,250	2.0769
venturiLevel3	4,026,819	8,054,237	672	2.0014	1,001,420	111,528	1,113,531	2.0613	351,929	**2.0613**
wikipedia-20051105	1,634,989	18,540,603	1,561	126.5925	14,248	418,588	434,379	**127.0162**	872,899	127.0162
wikipedia-20060925	2,983,494	35,048,116	3,643	138.7406	43,194	967,617	1,014,476	**140.5966**	1,919,124	140.5966
wikipedia-20061104	3,148,440	37,043,458	3,862	140.5598	47,102	1,031,432	1,082,416	**141.6711**	2,063,044	141.6711

Table 4.6: Results on instances in the *Large* bucket. All times are in milliseconds.

Graph Properties			*Greedy Peeling*		*Hybrid*				*Goldberg's*	
Instance	$\lvert V\rvert$	$\lvert E\rvert$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
europe_osm	50,912,018	54,054,660	6,869	1.7047	640	26	7,535	**2.0000**	–	–
hollywood-2009	1,139,905	56,375,711	1,895	**1,104.0000**	14,712	199,468	216,076	1,104.0000	–	–
kmer_U1a	67,716,231	69,389,281	26,907	4.0000	862	2	27,771	**4.0455**	–	–
kmer_V2a	55,042,369	58,608,800	20,570	6.9000	691	10	21,271	**7.0909**	–	–
rgg_n_2_23_s0	8,388,608	63,501,393	4,072	11.0476	112	25	4,210	**13.4000**	–	–
rgg_n_2_24_s0	16,777,216	132,557,200	8,571	12.1220	237	15	8,824	**13.7143**	–	–
road_usa	23,947,347	28,854,312	4,545	1.5974	301	2	4,849	**1.8462**	–	–
soc-orkut	4,847,571	106,349,209	9,111	**206.9307**	509	–	–	–	–	–

Table 4.7: Results on instances in the *Massive* bucket. All times are in milliseconds.

Graph Properties			*Greedy Peeling*		*Hybrid*				*Goldberg's*	
Instance	$\lvert V\rvert$	$\lvert E\rvert$	t_G	f_G	t_2	t_3	t_H	f_H	t_E	f^*
ca2010	710,145	1,744,683	856	**6,234,021.0000**	21	1	881	6,234,021.0000	103,815	6,234,021.0000
cond-mat-2003	31,163	120,029	16	**17.6000**	1	< 1	18	17.6000	3,032	17.6000
cond-mat-2005	40,421	175,693	23	**23.0000**	1	< 1	25	23.0000	4,836	23.0000
fl2010	484,481	1,173,147	496	3,753,682.4620	15	20	538	**3,992,056.5380**	59,637	3,992,056.5380
ga2010	291,086	709,028	257	**3,929,610.0000**	8	4	275	3,929,610.0000	33,232	3,929,610.0000
human_gene1	22,283	12,323,680	311	**62.6766**	26,139	142,612	169,065	62.6766	275,234	62.6766
il2010	451,554	1,082,232	444	**5,508,363.6000**	13	1	489	5,508,363.6000	57,320	5,508,363.6000
mi2010	329,885	789,045	299	6,993,878.8460	10	3	322	7,370,921.5830	39,088	**7,390,000.2310**
mo2010	343,565	828,284	321	**1,666,117.5000**	10	< 1	344	1,666,117.5000	41,163	1,666,117.5000
mawi_201512012345	18,571,154	19,020,160	9,216	798,116.4286	560	120	9,831	**927,951.0000**	–	–
mawi_201512020000	35,991,342	37,242,710	18,174	**1,770,103.0000**	1,073	183	19,219	1,770,103.0000	–	–
mouse_gene	45,101	14,461,095	419	27.7563	34,115	217,095	251,631	**28.4702**	505,157	28.4702
ny2010	350,169	854,772	328	2,986,674.1110	11	2	347	**3,289,936.6250**	42,839	3,289,936.6250
oh2010	365,344	884,120	344	**3,826,971.8000**	11	4	360	3,826,971.8000	43,112	3,826,971.8000
pa2010	421,545	1,029,231	420	**3,202,713.0000**	12	< 1	442	3,202,713.0000	52,870	3,202,713.0000
tx2010	914,231	2,228,136	1,265	6,563,105.3330	27	2	1,277	**6,630,141.8000**	120,507	6,630,141.8000

Table 4.8: Results on weighted instances. All times are in milliseconds.

5 Introduction to NDSR

Network design is a cornerstone of mathematical optimization, as witnessed by the large amount of literature on this topic. Indeed, historically it finds applications in logistics and transportation of goods and persons ([MW84]) and, more recently, in telecommunications, data sharing, energy distribution, and distributed computing ([GCF99]).

Network design is about defining the main characteristics of a network satisfying requirements on connectivity, capacity, and level-of-service. Setting up the network induces some installation cost, while additional costs are incurred when operating the service. It is quite common that a larger cost in the first term yields to a reduction in the latter, and vice-versa. Thus, the problem requires to find an equilibrium in the trade-offs between the installation and the operational costs.

A prototypical network design problem is the multi-commodity network design, in which one is required to design a network minimizing the installation cost of its arcs and the operational cost to serve a set of point-to-point connections, denoted as commodities. The solutions to this problem, however, can results in networks for which some commodities experience a low-quality connection with respect to some metric, e.g., distance or number of intermediate network nodes (hops) between origin and destination. In some applications, this is a critical issue: for example in telecommunications, a common requirement consists of limiting the number of hops between origin and destination of any connection, as this has a direct effect on the latency of the communication. Similarly, in transportation networks, it is common to limit the distance traveled between origin-destination pairs, in particular when dealing with a public transport service or when transporting perishable goods.

Recently, [BLM17] filled this gap and introduced the *Network Design with Service Requirements* (NDSR), a network design problem in which additional constraints impose that each origin-destination is served by a single path satisfying one or more level-of-service requirements. More specifically, each path must satisfy a maximum length with respect to a number of specified metrics. The problem asks to select some arcs to include in the network and to define, for each commodity, a path on the selected arcs and taking into account the mentioned level-of-service requirements. composed of The objective is to minimize a cost function consists of minimizing the total installation cost of the network arcs and of the operational cost of the selected paths. In that paper, the authors show that a model based on arc-flow variables can be hard to solve even for moderate-sized networks. Hence, through a wide polyhedral analysis they derive several families of valid inequalities, which can be exploited to strengthen the formulation. The resulting model, combined with an effective heuristic algorithm, allows to tackle larger instances of the problem.

In this manuscript, we propose a new model where variables are associated with paths satisfying the end-to-end service requirements. This way, many of the weaknesses of the arc-flow formulation are naturally overtaken without the need to recur to cut separation techniques. This desirable property comes at the cost of a formulation which is much larger, involving an exponential number of path variables. However, we show that for all the instances considered by [BLM17], we are indeed capable of quickly enumerating all the variables of the new formulation, thanks to an effective labelling algorithm, and to solve to proven optimality a much larger set of instances using a general-purpose ILP solver. In particular, our approach allows to solve a relevant fraction of the large instances introduced by [BLM17], and to compute near-optimal solutions in the remaining cases, showing that the algorithm scales efficiently to larger size of the network. In addition, we provide a new set of instances for which enumerating all the paths is not viable; for solving these large instances, we present a column generation scheme that is embedded into

a full branch-and-cut-and-price algorithm.

The following chapters in this dissertation are organized as follows. In the remainder of this section, we review some literature related to the problem at hand. Section 6 formally describes the problem, reviews a mathematical formulation from the literature, introduces a novel formulation, and compares the two models. Section 7.2 presents a solution approach based on branch-and-cut-and-price, describing column generation and the addition of valid inequalities. Section 8 computationally compares the performances of the proposed algorithms with state-of-the-art approach on test instances from the literature. Finally, in Section 9.2 we present some conclusions.

Literature Review: There is a wide literature on network design problems, and many surveys have been published on these topics, see, e.g., [MW84], [Cra00], and [Wie07]. Depending on the specific application, different variants of these problems were considered. A notable field of research involves the design of reliable and survivable networks, that has become a major objective for telecommunication operators (see, [KM05]). In this context, one is required to define a robust network preserving a given connectivity level under possible failure of certain network components. There exist several ways to express the network robustness. Under a stochastic paradigm, the network is required to remain operative either with a large probability ([SL13], [BCM15]) or after some recourse action has been implemented ([LMZ17]). Alternatively, more conservative approaches, imposing explicit redundancy in the definition of the network, have been considered in the literature; typically, one is required to design a network having two (edge) disjoint paths for each commodity ([MR05], [AS08], [ASK08], and [BMN09]), while [GMS95] considered the case in which higher connectivity requirements are imposed.

Another class of related problems arises in applications where explicit constraints are imposed on the characteristics of each path. A common requirement to guarantee the required quality of service is to limit the number of hops of each path; this problem has been introduced by

[BA92], while [Gou98] presented a strong flow formulation that has been later adopted for many hop-constrained network design problems. In some cases, the resulting network is required to have a special structure (typically, a tree), or survivability considerations have been added to the problem definition; see, e.g., [Bot+13] and [GLL15].

Our problem is closely related to the class of multi-commodity flow problems ([Ken78]) in which the network is given and commodities compete for the use of the arcs, which have a limited capacity. A branch-and-cut-and-price approach using path variables has been proposed by [BHV00]. Another relevant special case of NDSR arises when network design has to be defined for a unique commodity, and a single metric has to be considered. The resulting budget constrained shortest path problem, introduced by [Jok66], is an NP-hard problem, and turns out to be a simplified version of a subproblem that we have to solve for generating columns, which takes more than one metric into consideration.

Finally, on the applications side, end-to-end service requirements have been considered by [BS96], [Kim+99], and [ABW02], where express delivery of parcels is optimized. Though service time is a key aspect in these applications, the special structure of the networks allows to avoid to explicitly impose these constraints.

6 Problem description and formulation of NDSR

We now give a formal definition of the problem addressed in this paper. We are given a directed graph $G = (\mathcal{V}, \mathcal{A})$ where $\mathcal{V}$ is the node set and $\mathcal{A}$ is the arc set, and a set $\mathcal{K}$ of commodities. Each commodity $k \in \mathcal{K}$ has associated a source node s^k and a sink node t^k. For each arc $a \in \mathcal{A}$ there is an activation cost F_a; in addition, using an arc a for a commodity k induces a flow cost c_a^k. The problem asks to send, for each commodity k, one unit of flow on a single path p^k from the source to the sink, by determining a set of arcs and the routing of the flows so that the sum of the activation and flow costs is a minimum. In addition, there is a set $\mathcal{M}$ of metrics, that determines the feasibility of the path associated with a given commodity k: for each metric m, we denote by w_a^{km} the weight of arc a with respect to the metric, and require that the sum of the weights on arcs in p^k does not exceed a given upper limit W^{km}. We denote by $\mathbf{w}_a^k$ and $\mathbf{W}^k$ the corresponding m-dimensional vectors.

Throughout the paper, we assume that the graph includes no multiple arcs. This assumption is without loss of generality, as multiple arcs with different costs or service consumption for a given pair of nodes can be handled by the addition of dummy nodes. In addition, we assume that, for each commodity, at least one feasible path exists, since otherwise the problem is clearly infeasible.

The problem reduces to the budget-constrained shortest path when there is a single commodity and a single metric. This shows that the problem is NP-hard.

The next section reports a descriptive formulation that has been pro-

posed in the literature, whereas Section 6.0.2 introduces a novel formulation that will be used in our solution scheme.

6.0.1 Arc-flow formulation

The following formulation has been proposed by [BLM17] and makes use of activation variables and flow variables. All variables are binary and have the following meaning:

$$z_a = \begin{cases} 1 & \text{if arc } a \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \qquad (a \in \mathcal{A})$$

$$y_a^k = \begin{cases} 1 & \text{if commodity } k \text{ is routed on arc } a \\ 0 & \text{otherwise} \end{cases} \qquad (a \in \mathcal{A}, k \in \mathcal{K})$$

Then, the NDSR can be modelled using the following Integer Linear Programming (ILP) formulation:

$$\min \quad \sum_{a \in \mathcal{A}} F_a z_a + \sum_{k \in \mathcal{K}} \sum_{a \in \mathcal{A}} c_a^k y_a^k \tag{6.1a}$$

$$\text{subject to} \quad \sum_{a \in \delta^+(v)} y_a^k - \sum_{a \in \delta^-(v)} y_a^k = \begin{cases} +1 & v = s^k \\ -1 & v = t^k \\ 0 & v \in \mathcal{V} \setminus \{s^k, t^k\} \end{cases} \qquad k \in \mathcal{K} \tag{6.1b}$$

$$\sum_{a \in \mathcal{A}} w_a^{km} y_a^k \leq W^{km} \qquad k \in \mathcal{K}, m \in \mathcal{M} \tag{6.1c}$$

$$y_a^k \leq z_a \qquad a \in \mathcal{A}, k \in \mathcal{K} \tag{6.1d}$$

$$z_a \in \{0,1\} \qquad a \in \mathcal{A} \tag{6.1e}$$

$$y_a^k \in \{0,1\} \qquad a \in \mathcal{A}, k \in \mathcal{K}. \tag{6.1f}$$

The objective function minimizes the sum of the activation and flow costs. Constraints (6.1b) impose flow conservation for each commodity and node, whereas (6.1c) concern feasibility of the paths with respect to the metrics, and inequalities (6.1d) force the activation of arcs that are used for routing a positive flow. Finally (6.1e) and (6.1f) define the domain of the variables. The arc-flow formulation has a polynomial size, as it includes $(|\mathcal{K}| + 1)\,|\mathcal{A}|$ variables and $|\mathcal{K}|\,(|\mathcal{V}| + |\mathcal{A}| + |\mathcal{M}|)$ constraints.

6.0.2 Path-based formulation

The novel ILP formulation that we propose includes the same binary activation variables of model (6.1a)–(6.1f), that select the arcs to be activated, whereas flow variables are replaced by path variables that are defined as follows. Let $\mathcal{P}^k$ be the set of all feasible paths for commodity k. For each commodity k and each path $p \in \mathcal{P}^k$, let us introduce a binary path variable x_p with the following meaning:

$$x_p = \begin{cases} 1, & \text{if commodity } k \text{ is routed along path } p \\ 0 & \text{otherwise} \end{cases} \qquad (k \in \mathcal{K}, p \in \mathcal{P}^k)$$

Let c_p be the flow cost of the path p for commodity k, defined as the sum of the flow costs of all the arcs in p. The problem can thus be modelled

as follows:

$$\min \quad \sum_{a\in\mathcal{A}} F_a z_a + \sum_{k\in\mathcal{K}} \sum_{p\in\mathcal{P}^k} c_p x_p \tag{6.2a}$$

$$\text{subject to} \quad z_a - \sum_{p\in\mathcal{P}^k:\, a\in p} x_p \geq 0 \qquad a \in \mathcal{A}, k \in \mathcal{K} \tag{6.2b}$$

$$\sum_{p\in\mathcal{P}^k} x_p = 1 \qquad k \in \mathcal{K} \tag{6.2c}$$

$$z_a \in \{0,1\} \qquad a \in \mathcal{A} \tag{6.2d}$$

$$x_p \in \{0,1\} \qquad p \in \mathcal{P}^k, k \in \mathcal{K}. \tag{6.2e}$$

The objective function minimizes activation costs and flow costs, which are here expressed in terms of path variables. Constraints (6.2b) are the counterpart of (6.1d), enforcing activation of arcs that are used by a path. Constraints (6.2c) ensure that, for every commodity, one feasible path is selected. Finally, (6.2d) and (6.2e) define the domain of the variables.

Observation 1. *The model obtained by relaxing integrality requirement* (6.2e) *admits an optimal integer solution.*

Proof. Assume that an optimal solution for the relaxation is given. For a given choice of the z variables, the x variables associated with a commodity do not interact with those of a different commodity. Thus, we concentrate on a single commodity, say k, and assume that more than one path is selected for that commodity, the sum of the values of the associated path variables being 1. By optimality of the initial solution, all the selected paths must have the same cost. Hence, by increasing the value of one path variable to 1 and setting to 0 all the remaining ones, we obtain a solution that has the same cost as the original one. □

7 Algorithms to solve NDSR

7.1 Variable enumeration:

We first observe that the path-based formulation has has $O(2^{|\mathcal{V}|})$ variables and $(|\mathcal{A}|+1)\,|\mathcal{K}|$ constraints, i.e., its size can be exponential in the size of the instance. We now introduce an algorithm for enumerating all path variables; however, for large graphs, enumerating all paths can be challenging, and one may have to resort to column generation techniques, that will be discussed in Section 7.2.

Enumeration Algorithm 1 considers one commodity k at a time and defines all simple paths from s^k to t^k that satisfy resource constraints under all metrics. The algorithm is inspired by the labelling method proposed by [DB03] for the budget-constrained shortest path problem. In our algorithm, each label $\ell = \{u, c, \mathbf{w}\}$ represents a path from s^k to u having cost c and using w_m units of resources under each metric m. Each label is generated as unmarked, meaning that it has to be expanded, and then it is marked when considered for expansion. Expansion of a label ℓ associated with a node u consists in appending an arc $a = (u, v)$ to the current path. To this aim, we consider all the outgoing arcs from u and, for each neighbor node v not yet belonging to path ℓ, we check whether using the current label for reaching v preserves feasibility with respect to the metrics. In this case, we define a new label $\ell' = \{v, c + c_a^k, \mathbf{w} + \mathbf{w}_a^k\}$, i.e., we update the path cost and resource usage when using the current label for reaching v. Eventually, node v is inserted in set T, that includes all nodes associated with unmarked labels. The algorithm terminates when $T = \emptyset$, meaning that no label can be further expanded, and returns all labels associated with node t^k. Although a node can be inserted in

Algorithm 1: Compute all feasible paths for a fixed commodity

```
   Input   : k
1  s := s^k, t := t^k, T := {s}, c := 0, w := 0;
2  Define an unmarked label ℓ := {s, c, w} for node s;
3  while T ≠ ∅ do
4  |  pick any u ∈ T;
5  |  T := T \ {u};
6  |  foreach unmarked label ℓ = {u, c, w} associated with node u
   |   do
7  |  |  mark label ℓ;
8  |  |  foreach a = (u, v) ∈ δ⁺(u) do
9  |  |  |  if (v ∉ path ℓ) and (w + w_a^k ≤ W^k) then
10 |  |  |  |  define an unmarked label ℓ' = {v, c + c_a^k, w + w_a^k};
11 |  |  |  |  if (v ≠ t) then
12 |  |  |  |  |  T := T ∪ {v}
13 return all labels associated with node t;
```

and removed from T more than once, the convergence of the algorithm is ensured by requiring simple paths, which is checked in line 9.

The above algorithm can be improved by pre-computing, for each metric $m \in \mathcal{M}$, the shortest path from each node to t^k when the cost of each arc a is given by w_a^{km}. This figure can be used when checking feasibility of the new label in line 9: by adding this term to the left-hand-side of the inequality, we avoid generating labels that could not be feasibly expanded to node t^k.

7.1.1 Models comparison

In this section we compare the two formulations in terms of their linear relaxations.

Observation 2. *Any feasible solution for the linear relaxation of the path-based formulation can be mapped to a feasible solution of the same cost of the linear relaxation of the arc-based formulation, whereas the opposite does not hold.*

Proof. Let z^*, x^* be a feasible solution of the linear relaxation of the path-based formulation. We now define a solution $\widetilde{z}, \widetilde{y}$ that is feasible for the linear relaxation of the arc-based formulation and has the same cost. First, we set $\widetilde{z} = z^*$. Then, for each arc $a \in \mathcal{A}$ and commodity $k \in \mathcal{K}$, we set

$$\widetilde{y}^k_a = \sum_{p \in \mathcal{P}^k : a \in p} x^*_p.$$

It is straightforward to check that flow conservation constraints (6.1b) and feasibility requirements (6.1c) with respect to the metrics are satisfied as y variables are obtained as combination of feasible paths, whereas constraints (6.1d) are implied by (6.2c) and by the definition of $\widetilde{y}^k_a$. The equivalence of the costs follows from the definition of the cost of each path.

Figure 7.1 gives a small numerical example showing that the counterpart does not hold. The instance has no flow costs, a single commodity, and a single metric, for which the capacity is $W = 2$. For each arc we report the activation cost and the weight with respect to the metric. While there is a unique feasible path $p = \{(s, t)\}$ having cost 1, an optimal solution to the linear relaxation of the arc-based formulation is given by $y_{s1} = y_{1t} = y_{st} = 1/2$ having cost $1/2$. □

The observation shows that the path-based formulation dominates the arc-based one in terms of tightness of the associated linear relaxations.

The structure of feasible solutions for the linear relaxation of the arc-based formulations was analyzed by [BLM17], showing that fractional solutions may arise for two main reasons:

- for a given commodity, the model may route part of the flow on a path that is less expensive but infeasible with respect to the metric requirements (see again Figure 7.1);

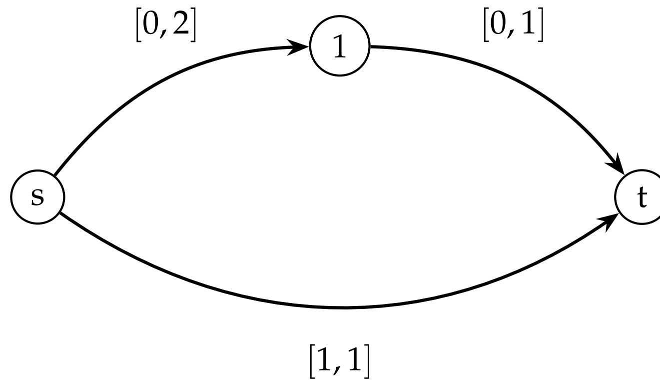

Figure 7.1: Simple example for which the path-based formulation dominates the arc-flow formulation.

- arc activation variables can be set at a fractional value to allow sharing the activation cost of some arcs among different paths associated with different commodities.

Accordingly, [BLM17] introduced different families of valid inequalities to cut some of these solutions. The first type of fractionalities do not appear in the path-based formulation, in which feasibility of the paths is enforced when defining the variables; thus, adding similar inequalities would be useless. On the other hand, the second type of fractionality may affect the path-based formulation as well, as shown in Figure 7.2. In this example, there are three commodities, no flow costs and activation costs equal to one for arcs $(3, 6), (4, 7), (5, 8)$ and zero for the remaining arcs. The figure shows an optimal solution of the linear relaxation of the path-based formulation, where the flow of each commodity is split into two paths, the costly arcs are activated at value 0.5 and the resulting cost is 3/2. On the other hand, any integer feasible solution has a cost at least equal to 2. For this reason, in our approach we consider the possibility to add some classes of valid inequalities of the second type.

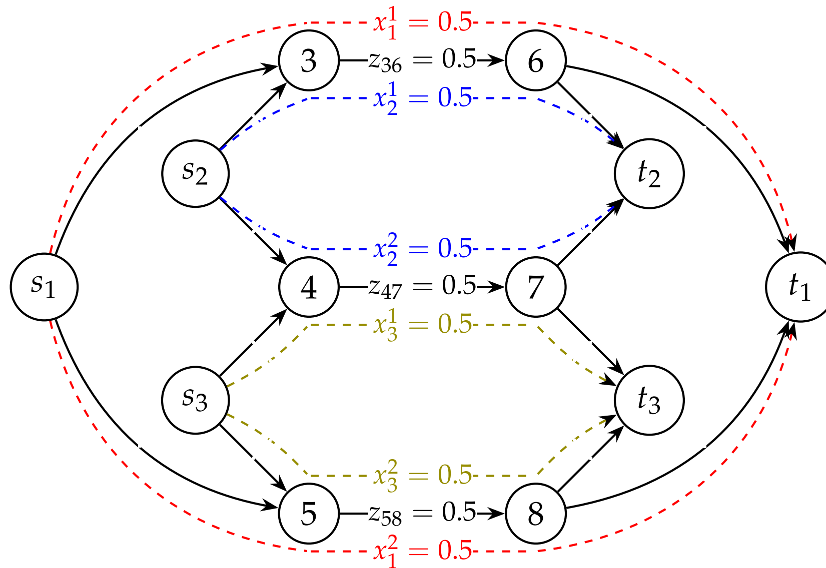

Figure 7.2: Fractional solution of the linear relaxation of the path-based formulation.

7.2 Branch-and-cut-and-price approach

In this section, we introduce an exact algorithm based on the path-formulation that can be used when enumerating all paths is unpractical. The algorithm adopts a branch-and-bound strategy and solves, at each node, the linear relaxation of the model by means of column generation techniques. The basic scheme is possibly enriched by the addition of valid inequalities, that do not change the structure of the method, thus resulting in a robust branch-and-cut-and-price algorithm.

7.2.1 Column generation and labelling

Column generation is an iterative scheme used for solving linear models with an exponentially large number of variables. At each iteration, a *restricted master problem* including a subset of the variables is solved, and

its dual solution is used to determine new variables (if any) that have to be added to the formulation in order to converge to an optimal solution.

In our setting we assume without loss of generality that constraints (6.2c) are rewritten as inequalities. At each iteration, the restricted master includes all the z variables, and a non-empty subset $\widetilde{\mathcal{P}^k} \subseteq \mathcal{P}^k$ of path variables for each commodity k (notice that by construction the restricted master always includes a feasible solution). Assume that the restricted master has been solved to optimality, and let γ_a^k and ρ^k be optimal non-negative dual variables associated with constraints (6.2b) and (6.2c), respectively. The *reduced cost* for a path variable x_p for a commodity k is given by

$$\bar{c}_p = c_p + \sum_{a \in p} \gamma_a^k - \rho^k = \sum_{a \in p} \left(c_a^k + \gamma_a^k\right) - \rho^k = \sum_{a \in p} \tilde{c}_a^k - \rho^k,$$

where the arc costs are $\tilde{c}_a^k = c_a^k + \gamma_a^k$. Thus, the *pricing problem* for a given commodity k is to find a feasible path whose reduced cost is negative, and can be formulated as a *budget-constrained shortest path problem* under costs $\tilde{c}_a^k$ and resources defined by the metrics. If the cost of this shortest path is strictly smaller than ρ^k, the corresponding path variable is added to the restricted master, and the process is iterated; if no path variable is generated for any commodity, the optimal solution of the current restricted master is an optimal solution for the linear relaxation of the problem.

Solution of the budget-constrained shortest path problem: Enumeration Algorithm 1 can be modified to compute the shortest path under resource constraints for a given commodity k, a problem which is NP-hard even if the graph is acyclic and $|\mathcal{M}| = 1$ (see, [GJ79]). The resulting Algorithm 2 differs from the enumeration one starting from line 11, where a dominance check aimed at avoiding expansion of suboptimal paths is introduced. More precisely, label ℓ' is dominated by another label ℓ'' associated with the same node if its cost and its resource usage are larger then or equal to the cost and usage of ℓ''. In this case ℓ' is

marked. Vice-versa, it may also happen that ℓ' dominates ℓ'', in which case we mark ℓ''. Node v is inserted in set T only if label ℓ' remains unmarked. The algorithm returns a unique path, corresponding to the label with minimum cost among all those associated with node t^k.

Algorithm 2: Compute a constrained shortest path for a fixed commodity

Input : k

1 $s := s^k, t := t^k, T := \{s\}, c := 0, \mathbf{w} := \mathbf{0}$;
2 Define an unmarked label $\ell := \{s, c, \mathbf{w}\}$ for node s;
3 **while** $T \neq \emptyset$ **do**
4 pick any $u \in T$;
5 $T := T \setminus \{u\}$;
6 **foreach** unmarked label $\ell = \{u, c, \boldsymbol{w}\}$ associated with node u **do**
7 mark label ℓ;
8 **foreach** $a = (u, v) \in \delta^+(u)$ **do**
9 **if** ($v \notin$ path ℓ) and ($\boldsymbol{w} + \boldsymbol{w}_a^k \leq \boldsymbol{W}^k$) **then**
10 define an unmarked label $\ell' = \{v, c + \tilde{c}_a^k, \mathbf{w} + \mathbf{w}_a^k\}$;
11 **if** (ℓ' is dominated by a label ℓ'' associated with node v) **then**
12 mark label ℓ';
13 **if** (ℓ' dominates a label ℓ'' associated with node v) **then**
14 mark label ℓ'';
15 **if** ($v \neq t$) and (ℓ' is unmarked) **then**
16 $T := T \cup \{v\}$

17 **return** the unmarked label with minimum cost c associated with node t;

7.2.2 Branching scheme

In our branching scheme we always select a z variable for branching. According to Observation 1, at each node where all the z variables attain integer values, there exists an optimal solution in which all the x variables are integer as well. Notice that this is the solution returned by solving the restricted master problem by means of the simplex algorithm.

A positive effect of this branching strategy is that it does not affect the structure of the pricing subproblem. This is a crucial property for designing an effective branch-and-price algorithm, as it allows to solve the column generation subproblem throughout all the branching tree by means of the same effective labelling algorithm used at the root node. Clearly, imposing $z_a = 1$ for some $a \in \mathcal{A}$ has no direct effect in the pricing. Conversely, when imposing $z_a = 0$, in the pricing subproblem we simply forbid the use of arc a when generating new path variables, which can be easily handled by setting $\mathcal{A} = \mathcal{A} \setminus \{a\}$.

7.2.3 Adding valid inequalities

In order to tighten the formulation and increase the dual bound at each node, we can add valid inequalities that cut fractional solutions in which arc activation variables are set at a fractional value to allow sharing the activation cost of some arcs among different paths.

To this aim, we adapt to our model some of the inequalities introduced by [BLM17] for the arc-flow formulation. These inequalities are obtained by analyzing the structure of the graph G and by deriving relationships between pairs of arcs (a, b) when routing the flow of a commodity k, namely:

- *OR* relationships, occurring when no more than one arc of pair (a, b) can be used to route flow from s^k to t^k;
- *IF* relationships, occurring when the flow through arc a must also be routed through b; and

- *CUT* relationship, occurring when at least one between a and b must be used to route the flow.

These relationships are then used to derive conditions that link the activation variable of an arc with the flow variables associated with the same arc and different commodities. By using the arc-flow variables, all these inequalities have the following general structure

$$\sum_{(a,k)\in C} z_a - \sum_{(a,k)\in C} y_a^k \geq q,$$

where C is a set of arc-commodity pairs and q is a scalar number.

By translating these conditions in terms of the path variables, we obtain

$$\sum_{(a,k)\in C} z_a - \sum_{(a,k)\in C} \sum_{p\in\mathcal{P}^k:a\in p} x_p \geq q \tag{7.1}$$

which can be enforced in the path-based formulation.

As it happens for the branching conditions, the addition of the inequalities above does not affect the structure of the pricing problem at a generic node of the branching tree. Indeed, for a given commodity k, constraint (7.1) only affects those paths that contain an arc a such that pair $(a,k) \in C$. For each such path, the reduced cost of the associated variable is thus $\bar{c}_p = \sum_{a\in p} \left(c_a^k + \gamma_a^k\right) + \phi^C - \rho^k$ where ϕ^C is the dual variable associated with constraint (7.1). More in general, given a collection $\mathcal{C}$ of inequalities, the reduced cost of a path associated with commodity k is

$$\bar{c}_p = \sum_{a\in p} \left(c_a^k + \gamma_a^k + \sum_{C\in\mathcal{C}:(a,k)\in C} \phi^C\right) - \rho^k$$

Hence, the only effect of additional inequalities on the shortest path computation is on the definition of arc costs $\tilde{c}_a^k$, which now include the dual variables of these constraints as well.

This allows us to solve the column generation subproblem with no modification of the labelling algorithm even after the addition of valid inequalities. The resulting algorithm is then a robust branch-and-cut-and-price.

8 Computational experiments

In our computational experiments we explore three directions. First, we compare the computational performance of the path-based formulation with the arc-based formulation. Our second order of business is to determine the features of the instances for which full enumeration of all feasible paths is possible, and when instead one has to resort to column generation. In this case, the solver cannot be used as a black box, and the addition of valid inequalities may be an effective option for accelerating the solution process. Finally, we evaluate the effect of adding valid inequalities to the path-based formulation, in terms of bound given by the linear relaxation and overall performance of the algorithm.

Unless specified, all algorithms were run on an AMD Ryzen Threadripper 3960X running at 3.8 GHz in single-thread mode, with a time limit of 1 hour per instance. All algorithms were implemented in C++. Both the arc-flow and the path-based formulations were solved using Gurobi version 9.1.1 as ILP solver, whereas the branch-and-cut-and-price was implemented on top of the SCIP optimization suite (version 7.0.1 with its default SoPlex solver), which allows to embed a column generation scheme within the enumeration process (see [Gam+20]).

8.1 Instances from the literature

We now describe a benchmark of instances that has recently been introduced by [BLM17], who kindly provided us the code for generating the numerical data. Each instance is characterized by the following parameters: the number of nodes $|\mathcal{V}|$, number of arcs $|\mathcal{A}|$, and number of

commodities $|\mathcal{K}|$. Nodes are randomly located on a rectangular grid and are connected by a spanning arborescence; then, $|\mathcal{A}| - |\mathcal{V}| + 1$ arcs are added to the arc set, making sure that the resulting network contains one directed path for each pair of nodes. The source and terminal node for each commodity are randomly selected in $\mathcal{V}$. The activation cost of each arc depends on the euclidean distance between the two endpoints and on a random parameter. A parameter γ governs the ratio between flow costs and activation costs. Coefficients $\mathbf{w}_a^k$ for a given arc $a \in \mathcal{A}$ are negatively correlated to the activation cost F_a through a parameter β and a random term. All instances consider $|\mathcal{M}| = 2$ metrics. Weight limits for each commodity k and each metric equal the length (using arc weights as lengths) of the q-th shortest path from s^k to t^k, where q is a random parameter having uniform distribution in an interval of size ΔQ centered in Q_{avg}. A particular combination of network size ($|\mathcal{V}|$, $|\mathcal{A}|$, and $|\mathcal{K}|$), cost structure and service requirements (β, γ, Q_{avg}, and ΔQ) is referred to as a scenario. Overall, [BLM17] defined 18 scenarios: the first seven scenarios share the same default values of the parameters for cost structure and service requirements, while considering varying network sizes ranging from 30 nodes, 120 arcs, and 90 commodities to 50 nodes, 250 arcs, and 150 commodities. Scenarios 8-15 are all defined with a fixed network size ($|\mathcal{V}| = 50$, $|\mathcal{A}| = 200$, and $|\mathcal{K}| = 150$) and different cost structure and service requirements. Finally, the last 3 scenarios have the default values of the parameters defining cost structure and service requirements and are characterized by larger size of the network, up to 80 nodes, 320 arcs, and 240 commodities. For each scenario, five instances were generated, for a total of 90 instances. Although the original set of instances is not available, we generated 18 scenarios for a total of 90 instances by using the same parameters used by [BLM17]. In Table 8.1, we will refer to each scenario as $|\mathcal{V}|/|\mathcal{A}|/||||/\beta\,\gamma\,Q_{avg}\,\Delta Q$ where the last four parameters take values in $\{L, M, H\}$ to denote low, medium and high figures, respectively.

8.2 Results on the instances from the literature

Table 8.1 gives the results of computational experiments on the 90 instances derived from the 18 scenarios described above; instances are grouped by scenario, i.e., every row reports aggregate results for five instances. The table compares the following approaches:

- base-model corresponds to the direct application of general-purpose ILP solver Gurobi to the arc-flow formulation;
- BLM is the *composite* algorithm proposed by [BLM17], and implements a branch-and-cut scheme built on top of the general-purpose ILP solver CPLEX 12.5.1 for solving the arc-flow formulation. The algorithm includes separation of several families of valid inequalities and an effective LP-based heuristic algorithm that is executed at the root node. All these figures are taken from [BLM17], and correspond to experiments executed on a Intel core i5 using an integrality gap for early termination equal to 0.1%;
- all-path denotes the algorithm obtained by enumerating all feasible paths through Algorithm 1 and solving the resulting path-based formulation using the Gurobi ILP solver. This approach does not include cut separation nor column generation, allowing us to use the solver as a black box, so as to exploit its full capabilities.

For each solution approach, the table reports the number of instances solved to proven optimality, the average percentage gap, and the average computing time (in seconds, with respect to instances that are solved to optimality only). For a given instance of the problem, let L and U be the best lower and upper bound, respectively, produced by an algorithm; the resulting percentage gap is computed as $100\frac{U-L}{U}$. For algorithm BLM, detailed computational results are only available for the instances of the first 7 scenarios. In addition observe that, for some scenarios, this algorithm solves all the associated 5 instances to optimality though returning a strictly positive percentage gap, due to the tolerance value

that is used within the algorithm. Finally, for algorithm all-path we also report the number of path variables enumerated by the labelling algorithm. The enumeration time is always very small (at most 0.5 seconds) and it is included in the computing time of the algorithm.

		base-model			BLM ([BLM17])			all-path			
	scenario	# opt	% gap	time	# opt	% gap	time	# opt	% gap	time	# path
1	30/120/90/MMMM	5	0.00	446.75	5	0.02	175	5	0.00	1.41	895
2	40/160/120/MMMM	4	0.21	1352.68	4	0.18	133	5	0.00	7.66	1098
3	50/150/150/MMMM	5	0.00	776.90	5	0.08	230	5	0.00	3.24	1409
4	50/200/100/MMMM	2	1.36	2953.33	4	0.43	1055	5	0.00	41.71	956
5	50/200/150/MMMM	1	5.93	2817.12	2	1.33	350	5	0.00	478.80	1455
6	50/200/200/MMMM	0	3.81	–	3	0.45	735	5	0.00	324.22	1898
7	50/250/150/MMMM	0	9.20	–	1	2.61	2631	4	0.38	171.85	1388
8	50/200/150/LMMM	2	3.09	2455.09		0.50		5	0.00	32.90	1249
9	50/200/150/HMMM	0	12.02	–		2.00		3	1.64	365.97	1795
10	50/200/150/MLMM	0	9.02	–		0.90		5	0.00	639.56	1455
11	50/200/150/MHMM	1	6.60	3557.67		0.10		5	0.00	782.21	1455
12	50/200/150/MMLM	5	0.00	1009.59		0.10		5	0.00	1.51	804
13	50/200/150/MMHM	0	10.68	–		1.90		4	0.64	345.37	2085
14	50/200/150/MMML	0	6.26	–		0.70		5	0.00	305.79	1421
15	50/200/150/MMMH	1	2.79	1902.54		0.70		5	0.00	100.74	1153
16	60/240/180/MMMM	0	10.68	–		0.70		5	0.00	603.16	1711
17	70/280/210/MMMM	0	11.91	–		2.50		2	0.54	686.22	1961
18	80/320/240/MMMM	0	15.86	–		2.40		2	1.25	951.94	2307
	summary	26	6.08	1371.97	45*	0.98		80	0.25	288.22	1472

Table 8.1: Results on instances from the literature.

The results confirm the outcome of the computational experiments reported by [BLM17] for the first seven scenarios, i.e., that algorithm BLM outperforms the base-model, which can solve only small instances and has large percentage gaps for most unsolved scenarios. Instead, results borrowed from [BLM17] show that the addition of valid inequalities and the use of an effective heuristic yields to an algorithm which is able to

solve 24 instances out of 35, with average percentage gap equal to 0.73. Both these approaches are dominated by algorithm all-path, which solves all but one instances in the first seven scenarios, and has a percentage gap equal to 0.38 for the remaining instance. This is due to the fact that the formulation is tight and that, for these instances, the number of path variables does not grow up: this number is always smaller than 2000, which makes the model solvable with a limited computational effort. All instances for scenarios 1 and 3 are solved by both BLM and all-path: for these scenarios, the average computing time of the former is two orders of magnitude slower than the latter (although BLM was executed on a slightly slower machine and used a different ILP solver).

For what concerns the instances in scenarios 8-15, the performances of algorithm all-path remain satisfactory. The algorithm solves 37 of the 40 associated instances, and has an average percentage gap equal to 0.28. Finally, for very large instances (scenarios 16 to 18), the algorithm solves 9 instances out of 15 and has an average percentage gap equal to 0.60. Overall, our algorithm solves to proven optimality almost 90% of the instances with an average percentage gap of 0.25. [BLM17] do not report detailed results for all scenarios, but instead mention that BLM only solves 45 instances (for this reason this figure is marked with an asterisk in the summary line of the table) and has an average percentage gap of 0.98.

8.3 Results on additional instances

The results in Table 8.1 show that, for the instances from the literature, the number of feasible paths is quite small. Thus, not surprisingly, the all-path approach is always better than base-model and BLM. Our second set of experiments is aimed at evaluating the limits of applicability of explicit enumeration of all path variables, and the alternative use of the branch-and-price algorithm described in Section 7.2 when enumeration is unpractical. Hence, we generated additional instances derived from

the instances in scenarios 1–7, in which the number of feasible paths is increasing. To minimize the number of parameters for defining the additional instances, we simply introduce a parameter $\alpha \geq 1$ that is used to scale each upper limit W^{km} for a commodity k and metric m. This has the effect to make less binding the constraints defining the feasibility of a path with respect to the metrics.

Table 8.2 reports aggregated results, summarizing 35 instances per line, obtained with different values of α ranging from 1.00 to 3.00. We compare the base-model, the all-path approach, and the branch-and-price algorithm and report, for each solution method, the number of optimal solutions, the average percentage gap and the average computing time (with respect to instances solved to optimality only). For all-path we also report the total number of feasible paths; this figure is averaged over all the 35 instances of a line, provided that enumeration of all paths was completed within the time limit for all the instances. Finally, for branch-and-price we give the average number of path variables that have been generated during the execution of the algorithm (with respect to instances solved to optimality only).

	base-model			all-path				branch-and-price			
α	# opt	% gap	time	# opt	% gap	time	# path	# opt	% gap	time	# path
1.00	17	2.93	1191.34	34	0.05	146.25	1300	31	0.38	401.20	1116
1.25	3	8.45	1680.47	21	1.45	410.74	6428	16	2.65	816.47	5896
1.50	6	6.01	976.83	20	1.72	514.40	33,178	14	2.64	667.09	10,816
1.75	9	3.97	903.57	19	1.71	480.65	169,286	17	2.15	539.52	11,209
2.00	14	2.40	798.33	17	1.97	449.87	855,441	21	1.55	830.85	10,110
2.25	18	1.53	613.12	14	–	673.01	–	23	1.19	522.94	9115
2.50	22	0.91	654.06	9	–	957.18	–	26	0.80	475.83	7743
2.75	23	0.70	554.46	5	–	998.54	–	27	0.68	577.48	7479
3.00	25	0.61	470.26	3	–	1627.80	–	28	0.58	628.28	7048

Table 8.2: Results on additional instances.

The results in Table 8.2 show that, for values of $\alpha < 2$, the total number of paths is still manageable (below 200,000) and all-path remains the best option. Conversely, for larger values of α, in many cases enumerating all path variables within the time limit is not possible or the path-based formulation has too many variables, and hence a method based on column generation is advisable. Indeed, for $\alpha = 2$, branch-and-price solves 21 instances compared to the 17 solved by all-path, and this gap increases for larger values of α. Finally, we observe that the performances of the base-model as well improve for increasing α, which suggests that the problem is easier when feasibility constraints are not too demanding. This confirms the outcome of some observations by [BLM17] about the structure of optimal solutions of the linear relaxation of this formulation, as these solutions are allowed to use infeasible paths at a fractional level.

8.4 Strengthening the model

As already mentioned, the BLM approach is based on a branch-and-cut algorithm in which the arc-flow formulation is iteratively strengthened by means of valid inequalities, designed to cut off infeasible solutions of the linear relaxation. [BLM17] showed that adding these inequalities is beneficial to the algorithm, in terms of value of the dual bound at the root node and number of instances that can be solved to optimality.

Our third set of experiments is thus aimed at evaluating the impact of adding valid inequalities to the path-based formulation. Table 8.3 gives the outcome of our experiments on instances in scenarios 1–7 for the branch-and-price approach without and with the addition of valid inequalities (branch-and-cut-and-price).

The table is organized in two parts. In the first one, we report the average percentage gap of the linear relaxation in the two configurations, and the associated computing time reported by SCIP. For the version of the algorithm with cuts, we borrowed from [BLM17] the following families of inequalities: 3OR, 1CUT-IF and 1OR-IF, obtained by com-

	linear relaxation				exact solution					
	without cuts		with cuts		branch-and-price			branch-and-cut-and-price		
scenario	% gap	time	% gap	time	# opt	% gap	time	# opt	% gap	time
1	5.38	0.26	3.22	31.39	5	0.00	17.90	5	0.00	52.00
2	4.77	0.48	2.43	102.27	5	0.00	63.27	5	0.00	164.91
3	2.79	0.49	1.24	102.82	5	0.00	46.46	5	0.00	146.63
4	6.11	0.70	3.79	186.62	5	0.00	380.83	5	0.00	454.94
5	6.58	1.52	4.00	286.55	3	1.11	220.35	3	0.98	419.83
6	5.02	1.60	2.64	357.34	4	0.58	250.27	4	0.48	549.92
7	7.31	2.25	4.37	600.73	4	0.99	2058.17	4	0.78	1702.62
summary	5.42	1.04	3.10	238.25	31	0.38	401.20	31	0.32	463.29

Table 8.3: Results on the addition of valid inequalities.

bining three OR conditions, one CUT with one or more IF conditions, and one OR with one or more IF conditions, respectively. The reader is referred to [BLM17] for the definition of these inequalities as well as to their separation; additional inequalities from this paper showed to have a very marginal effect in our preliminary computational experiments. Separation is carried out at the root node until no violated cut is found, according to SCIP tolerance. The results in Table 8.3 confirm that the addition of valid inequalities produces a tighter formulation for which the dual gap with respect to the optimum value is quite small, and reduced by 42% with respect to the formulation without cuts (from 5.42% to 3.10%). However, separating these inequalities is time consuming in practice, which prevents the exhaustive separation of the cuts in an enumerative approach.

For this reason, in the rightmost part of the table we consider a branch-and-cut-and-price algorithm, in which separation is embedded into the branch-and-price in a heuristic way as follows: cuts are added at the root node only, and at most 25 rounds of separation are performed. At each separation round, we consider in order 3OR, 1CUT-IF and 1OR-IF,

and we stop the separation as soon as a valid inequality is obtained. The inequality is added to the restricted master problem which is then re-optimized. This heuristic approach is justified by some preliminary experiments on each family of inequalities, where we evaluated the computational effort required for deriving a valid inequality and the relative effect of the inequality on the dual bound. Remind that a nice property of our approach is that the addition of new cuts does not affect the structure of the pricing subproblem, yielding a robust branch-and-cut-and-price approach. For both branch-and-price and branch-and-cut-and-price we report the number of optimal solutions, the average percentage gap and the average computing time.

The results on the exact methods show that branch-and-cut-and-price solves the same number of instances as branch-and-price, and produces slightly better gaps for unsolved instances. Indeed, both algorithms solve 31 instances, the average percentage gaps being 0.38 (for branch-and-price) and 0.32 (for branch-and-cut-and-price). Despite adding valid cuts seems to be very effective in closing the gap at the root node, its limited contribution within an enumerative scheme is due to the computational overhead required for separating cuts and for solving larger models at each decision node.

9 Conclusions and Future Work

9.1 Dense Subgraphs

In the fist part of the dissertation, we studied a non-linear graph optimization problem that requires to determine the densest subgraph in a given graph. While some prior work mentioned that the so-called *Greedy Peeling* algorithm is good in practice, we concluded empirically that *Greedy Peeling* finds dense subgraphs which are close to the optimal subgraphs across a range of graph sizes. We provided a simple connected instance for which the greedy algorithm shows its worst case performance. We introduced a new heuristic algorithm that combines this fast and effective greedy algorithm and an exact method from the literature. The extensive experiments done to measure the performance of this new heuristic suggest that, for a sizeable number of real world instances, we can improve upon the solution provided by the *Greedy Peeling* using our new heuristic. We have presented an efficient implementation of the algorithms to solve both unweighted and weighted instances, with the aim of attacking instances of very large size, like those arising, e.g., in social network applications. To the best of our knowledge, this is the most comprehensive computational study on DSE problem involving instances with tens of millions of vertices and hundreds of millions of edges.

9.2 NDSR

In the later part of the dissertation, we considered an NP-hard network design problem with end-to-end service requirements that play a fundamental role in many contexts, including telecommunications and transportation. From a modelling viewpoint, we proposed a novel ILP formulation in which variables are associated with feasible paths, and discussed alternative ways for handling the exponential number of variables in the model. From a methodological perspective, we showed how a column generation algorithm can be embedded into a branch-and-cut-and-price scheme, that is robust in the sense that the structure of the subproblems is not altered by the branching conditions nor by the addition of valid inequalities. Finally, we gave a comprehensive computational analysis of the performances of the proposed algorithm, which is compared with a state-of-the-art approach proposed in the recent literature. Our computational experiments showed that the proposed algorithm outperforms its competitor and scales efficiently to larger size of the network.

The introduced path-based formulation is quite general, as all the nasty constraints appear in the definition of feasible paths only. For this reason, it may be worthy to use this modelling approach for other multi-commodity network design problems arising in different contexts.

9.2.1 Application of NDSR to Hop Constrained Survivable Network

One potential application of NDSR is to solve Survivable Network Design problems. [BMN09] describes Survivable Network Design (SND) as a network design problem that tries to minimize the cost associated with the network while ensuring that there are a required minimum number of paths that are edge disjoint. In [BMN09] , the authors describe SND as selecting the edges in a network which minimizes the total cost while also meeting the connectivity requirements — namely number of edge

disjoint paths between the desired nodes. For a more comprehensive discussion of SND, reader can refer to the aforementioned [BMN09]

[Bot+13] discusses HOP-SND as a special case of SND where the number of hops that the feasible paths can take are limited. While SND ensures the necessary level of edge disjoint paths to protect against link failures but this might lead to the case where the optimal solution can have paths which have too many hops and hence can lead to prohibitive delays. By introducing the hop constraint, we can eliminate these delays.

Both SND and HOP-SND can potentially be solved using the algorithms developed in the previous problems to solve the NDSR problems. We are currently modfiying the Algorithms 1 and 2 to solve the HOP-SND problem.

Bibliography

[ABW02] A. P. Armacost, C. Barnhart, and K. A. Ware. "Composite Variable Formulations for Express Shipment Service Network Design." In: *Transportation Science* 36.1 (2002), pp. 1–20.

[AC09] R. Andersen and K. Chellapilla. "Finding Dense Subgraphs with Size Bounds." In: *Algorithms and Models for the Web-Graph*. Ed. by K. Avrachenkov, D. Donato, and N. Litvak. Berlin, Heidelberg: Springer, 2009, pp. 27–37.

[ARS02] J. Abello, M. Resende, and S. Sudarsky. "Massive Quasi-Clique Detection." In: *LATIN 2002: Theoretical Informatics*. Ed. by S. Rajsbaum. Berlin, Heidelberg: Springer, 2002, pp. 598–612.

[AS08] A. K. Andreas and J. C. Smith. "Mathematical Programming Algorithms for Two-Path Routing Problems with Reliability Considerations." In: *INFORMS J. Comput.* 20.4 (2008), pp. 553–564.

[Asa+00] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. "Greedily Finding a Dense Subgraph." In: *Journal of Algorithms* 34.2 (2000), pp. 203–221.

[ASK08] A. K. Andreas, J. C. Smith, and S. Küçükyavuz. "Branch-and-price-and-cut algorithms for solving the reliable h -paths problem." In: *J. Glob. Optim.* 42.4 (2008), pp. 443–466.

[BA92] A. Balakrishnan and K. Altinkemer. "Using a Hop-Constrained Model to Generate Alternative Communication Network Design." In: *ORSA Journal on Computing* 4.2 (1992), pp. 192–205.

[BBH11] B. Balasundaram, S. Butenko, and I. Hicks. "Clique Relaxations in Social Network Analysis: The Maximum *k*-Plex Problem." In: *Operations Research* 59.1 (2011), pp. 133–142.

[BCM15] J. Barrera, H. Cancela, and E. Moreno. "Topological optimization of reliable networks under dependent failures." In: *Operations Research Letters* 43.2 (2015), pp. 132–136.

[BHV00] C. Barnhart, C. A. Hane, and P. H. Vance. "Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems." In: *Operations Research* 48.2 (2000), pp. 318–326.

[BKV12] B. Bahmani, R. Kumar, and S. Vassilvitskii. "Densest Subgraph in Streaming and MapReduce." In: *Proc. VLDB Endow.* 5.5 (Jan. 2012), pp. 454–465. ISSN: 2150-8097. DOI: 10.14778/2140436.2140442.

[BLM17] A. Balakrishnan, G. Li, and P. Mirchandani. "Optimal network design with end-to-end service requirements." In: *Operations Research* 65.3 (2017), pp. 729–750.

[BMN09] A. Balakrishnan, P. Mirchandani, and H. P. Natarajan. "Connectivity Upgrade Models for Survivable Network Design." In: *Oper. Res.* 57.1 (Jan. 2009), pp. 170–186.

[Bol98] B. Bollobás. *Modern Graph Theory*. 1st ed. Graduate Texts in Mathematics 184. Springer-Verlag New York, 1998. ISBN: 978-0-387-98488-9,978-1-4612-0619-4.

[Bot+13] Q. Botton, B. Fortz, L. Gouveia, and M. Poss. "Benders Decomposition for the Hop-Constrained Survivable Network Design Problem." In: *INFORMS Journal on Computing* 25.1 (2013), pp. 13–26.

[BS96] C. Barnhart and R. R. Schneur. "Air Network Design for Express Shipment Service." In: *Operations Research* 44.6 (1996), pp. 852–863.

[Cha00] M. Charikar. "Greedy Approximation Algorithms for Finding Dense Components in a Graph." In: *Approximation Algorithms for Combinatorial Optimization*. Ed. by K. Jansen and S. Khuller. Berlin, Heidelberg: Springer, 2000, pp. 84–95.

[Cra00] T. Crainic. "Service network design in freight transportation." In: *European Journal of Operational Research* 122.2 (2000), pp. 272–288.

[CS12] J. Chen and Y. Saad. "Dense Subgraph Extraction with Application to Community Detection." In: *IEEE Transactions on Knowledge and Data Engineering* 24.7 (2012), pp. 1216–1230.

[DB03] I. Dumitrescu and N. Boland. "Improved Preprocessing, Labeling and Scaling Algorithms for the Weight-Constrained Shortest Path Problem." In: *Networks* 42.3 (2003), pp. 135–153.

[DH11] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection." In: *ACM Transactions on Mathematical Software* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[For10] S. Fortunato. "Community detection in graphs." In: *Physics Reports* 486.3–5 (2010), pp. 75–174.

[Fre78] L. C. Freeman. "Centrality in social networks conceptual clarification." In: *Social Networks* 1.3 (1978), pp. 215–239. ISSN: 0378-8733. DOI: https://doi.org/10.1016/0378-8733(78)90021-7.

[Gam+20] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig. *The SCIP Optimization Suite 7.0*. ZIB-Report 20-10. Zuse Institute Berlin, Mar. 2020.

[GCF99] B. Gendron, T. Crainic, and A. Frangioni. "Multicommodity Capacitated Network Design." In: *Telecommunications Network Planning*. Ed. by B. Sansò and P. Soriano. Springer Science & Business Media, 1999.

[GGT89] G. Gallo, M. Grigoriadis, and R. Tarjan. "A Fast Parametric Maximum Flow Algorithm and Applications." In: *SIAM Journal on Computing* 18.1 (1989), pp. 30–55.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. San Francisco: W.H. Freeman, 1979.

[GLL15] L. Gouveia, M. Leitner, and I. Ljubić. "The two-level diameter constrained spanning tree problem." In: *Mathematical Programming* 150 (2015), pp. 49–78.

[GMS95] M. Grötschel, C. L. Monma, and M. Stoer. "Polyhedral and Computational Investigations for Designing Communication Networks with High Survivability Requirements." In: *Operations Research* 43.6 (1995), pp. 1012–1024.

[Gol84] A. V. Goldberg. *Finding a Maximum Density Subgraph*. Tech. rep. Berkeley, CA, USA: University of California at Berkeley, 1984.

[Gou98] L. Gouveia. "Using Variable Redefinition for Computing Lower Bounds for Minimum Spanning and Steiner Trees with Hop Constraints." In: *INFORMS Journal on Computing* 10.2 (1998), pp. 180–188.

[GT15] A. Gionis and C. Tsourakakis. *Dense Subgraph Discovery (DSD)*. `http://people.seas.harvard.edu/~babis/dsd.pdf`. Accessed: 2019-10-16. 2015.

[GT88] A. Goldberg and R. Tarjan. "A New Approach to the Maximum-flow Problem." In: *Journal of the ACM* 35.4 (1988), pp. 921–940.

[Hen+18] M. Henzinger, A. Noe, C. Schulz, and D. Strash. "Practical Minimum Cut Algorithms." In: *Journal of Experimental Algorithmics* 23 (2018), 1.8:1–1.8:22.

[HI18] T. Hegeman and A. Iosup. "Survey of Graph Analysis Applications." In: *ArXiv* abs/1807.00382 (2018).

[HS18] Y. Hiroki and H. Satoshi. "Discounted average degree density metric and new algorithms for the densest subgraph problem." In: *Networks* 71.1 (2018), pp. 3–15.

[Jok66] H. Joksch. "The shortest route problem with constraints." In: *Journal of Mathematical Analysis and Applications* 14 (1966), pp. 191–197.

[Ken78] J. Kennington. "A survey of linear cost multicommodity network fows." In: *Operations Research* 26.2 (1978), pp. 209–236.

[Kim+99] D. Kim, C. Barnhart, K. Ware, and G. Reinhardt. "Multimodal Express Package Delivery: A Service Network Design Application." In: *Transportation Science* 33.4 (1999), pp. 391–407.

[KM05] H. Kerivin and A. R. Mahjoub. "Design of Survivable Networks: A survey." In: *Networks* 46.1 (2005), pp. 1–21.

[KS09] S. Khuller and B. Saha. "On finding dense subgraphs." In: *International Colloquium on Automata, Languages, and Programming*. Ed. by S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas. Berlin, Heidelberg: Springer, 2009, pp. 597–608.

[Lee+10] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. "A Survey of Algorithms for Dense Subgraph Discovery." In: *Managing and Mining Graph Data*. Boston, MA: Springer, 2010, pp. 303–336.

[LMZ17] I. Ljubić, P. Mutzel, and B. Zey. "Stochastic survivable network design problems: Theory and practice." In: *European Journal of Operational Research* 256.2 (2017), pp. 333–348.

[MR05] T. L. Magnanti and S. Raghavan. "Strong formulations for network design problems with connectivity requirements." In: *Networks* 45.2 (2005), pp. 61–79.

[MW84] T. L. Magnanti and R. T. Wong. "Network Design and Transportation Planning: Models and Algorithms." In: *Transportation Science* 18.1 (1984), pp. 1–55.

[PQ82] J.-C. Picard and M. Queyranne. "A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory." In: *Networks* 12.2 (1982), pp. 141–159.

[RKF12] S. U. Rehman, A. U. Khan, and S. Fong. "Graph mining: A survey of graph mining techniques." In: *Seventh International Conference on Digital Information Management (ICDIM 2012)*. 2012, pp. 88–92. DOI: 10.1109/ICDIM.2012.6360146.

[SL13] Y. Song and J. R. Luedtke. "Branch-and-cut approaches for chance-constrained formulations of reliable network design problems." In: *Mathematical Programming Computation* 5.4 (2013), pp. 397–432.

[Tso+13] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. "Denser Than the Densest Subgraph: Extracting Optimal Quasi-cliques with Quality Guarantees." In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Ed. by R. Ghani, T. Senator, P. Bradley, R. Parekh, and J. He. New York, NY, USA: ACM, 2013, pp. 104–112.

[Tso14] C. Tsourakakis. *A novel approach to finding nearcliques: The triangle-densest subgraph problem*. Tech. rep. ICERM, Brown University, 2014.

[Wie07] N. Wieberneit. “Service network design for freight transportation: a review.” In: *OR Spectrum* 30.1 (2007), pp. 77–112.