2016

On a 2-path problem

Song, Haotian

http://hdl.handle.net/11023/3314

Downloaded from PRISM Repository, University of Calgary
On a 2-Path Problem

by

Haotian Song

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN MATHEMATICS AND STATISTICS

CALGARY, ALBERTA
September, 2016

© Haotian Song 2016
Abstract

An electric power supplier needs to build a transmission line between 2 jurisdictions. Ideally, the design of the new electric power line would be such that it optimizes some user-defined utility function, for example, minimizes the construction cost or the environmental impact. Due to reliability considerations, the power line developer has to install not just one, but two transmission lines, separated by a certain distance from one to another, so that even if one of the lines fails, the end user will still receive electricity along the second line. We discuss how such a problem can be modelled and prove the general graph-based problem to be $\mathcal{NP}$-hard. At the same time, we propose a polynomial-time approximation scheme to handle this problem. Although the worst-case performance of the latter scheme is not fully understood yet, we note that under two mild practical assumptions, the scheme yields the optimal solution to the original problem. The novel scheme appears to be extremely efficient numerically. Our implementation scheme vastly outperforms more conventional solution methods, such as mixed-integer based models. In turn, this allows us to solve realistically sized problems on graphs nearing a hundred thousand nodes.
Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Yuriy Zinchenko, for his infinite support on everything.

The faculty at the Department of Mathematics and Statistics, University of Calgary, made this a terrific place to be. I would especially like to thank Prof. Alex Brudnyi, Prof. Cristian Rios, Prof. Karen Seyffarth, and Prof. Deniz Sezer from whose lovely and interesting courses I gained not only the beauty of mathematical knowledge but also the passion to pursue my study further.

I would like to thank Prof. Yves Lucet and Prof. Warren Hare from whom I took a course respectively at the University of British Columbia, Okanagan, during my stay in the winter semester of 2015. They astonished me with how active, prolific, and strong an Operations Research group could ever be.

I would also like to thank my office roommates Muhammad Ali Khan and Melanie Foerster who make our office another home for me.
To my parents

and all the time and chances I have ever wasted...
# Table of contents

Abstract ii
Acknowledgements iii
Dedication iv
Table of contents v
List of figures vii
List of tables viii
List of symbols ix

1 Background 1
   1.1 Introduction ................................................. 1
   1.2 Literature review ............................................. 4
   1.3 Motivational example ......................................... 6

2 Problem formulations 9
   2.1 Graph-based model ............................................. 9
       2.1.1 Real-world scenario .................................... 12
   2.2 0-1 integer programming model ................................ 14
       2.2.1 Min-cost-flow-type model ............................... 14
       2.2.2 Real-world scenario (continued) ....................... 17

3 Basic properties of DS2P models 19
   3.1 Graph-based model’s complexity ............................. 19
       3.1.1 Basic concepts review .................................. 20
       3.1.2 Hardness classification .................................. 23
   3.2 LP relaxation of the 0-1 programming model .................. 29

4 A novel approximation scheme 32
   4.1 3D Embedding .................................................. 32
       4.1.1 Assumptions yielding optimality ....................... 32
       4.1.2 Procedure ................................................ 33
4.2 Preliminary models' performance and discussion .......................... 37
4.3 Advanced implementation for large-scaled graph ......................... 40

5 Conclusion and future work 43

Bibliography 45

A MATLAB codes 49
   A.1 DiamondAdj.m .................................................. 49
   A.2 path3D.m ....................................................... 55
   A.3 compare3D.m .................................................. 79
# List of figures

1.1 An example of a shortest path. .................................................. 3  
1.2 Cost color-wash map. .............................................................. 6  
1.3 Hypothetical Northeastern Alberta power-line topology. .......... 7  
2.1 Illustration of Problem 2.3 ...................................................... 11  
2.2 A $K_{3,3}$ directed graph .......................................................... 11  
2.3 3-by-3 diamond directed graph ............................................... 12  
2.4 Graph with vertical edges ....................................................... 13  
2.5 “Curved” path segments .......................................................... 14  
2.6 3-by-3 diamond directed graph with a spurious sink-to-source edge . 18  
3.1 Relation of $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete and $\mathcal{NP}$-hard .... 22  
3.2 Digraph modification for $\Delta = 1$. ........................................ 24  
3.3 3SAT reduction subgraphs ...................................................... 25  
3.4 The graph constructed for $(x_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_1 \lor x_4) \land (\bar{x}_2 \lor x_3 \lor \bar{x}_4)$ .. 26  
3.5 The graph constructed for $(x_1 \lor x_2 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2 \lor \bar{x}_2) \land (\bar{x}_1 \lor x_2 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2 \lor x_2)$ 28  
3.6 Fractional-flow example .......................................................... 30  
4.1 Demonstration of 3D embedding ............................................. 33  
4.2 2D map .............................................................................. 34  
4.3 3D embedding .................................................................. 35  
4.4 Single shortest path in 3D ....................................................... 36  
4.5 Projections on 2D maps .......................................................... 36  
4.6 Two paths in 2D ................................................................. 37  
4.7 Optimal disjoint 2-path pair (a) against 2 individual shortest paths (b) . . 39  
4.8 Layer 2 & Layer 3 ................................................................. 41
List of tables

4.1 Average run-time in seconds for the two schemes; for the 0-1 IP scheme we
only solve its root LP relaxation. .......................... 38
4.2 Average run-time in seconds for 3D embedding: naïve vs layered approach . 41
## List of symbols

<table>
<thead>
<tr>
<th>Symbol or abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>A graph</td>
</tr>
<tr>
<td>$\mathcal{V}$</td>
<td>The vertex set of $G$</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>The edge set of $G$</td>
</tr>
<tr>
<td>$s$</td>
<td>The source of $G$</td>
</tr>
<tr>
<td>$t$</td>
<td>The sink of $G$</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>The number of disjoint paths</td>
</tr>
<tr>
<td>${1,2,...,\eta}$</td>
<td>Vertex indices</td>
</tr>
<tr>
<td>$(i,j)$</td>
<td>An arc (edge) from vertex $i$ to $j$</td>
</tr>
<tr>
<td>$c_{ij}$</td>
<td>The edge weight of $(i,j)$-edge</td>
</tr>
<tr>
<td>$\langle u,v \rangle$</td>
<td>A path from vertex $u$ to $v$ in a directed graph or a path between vertex $u$ and $v$ in an undirected graph</td>
</tr>
<tr>
<td>$N_s,N_t$</td>
<td>Neighborhoods of $s$ and $t$</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>A fixed distance threshold</td>
</tr>
<tr>
<td>$K$</td>
<td>The size of the diamond-shaped digraph</td>
</tr>
<tr>
<td>$u_{ij}$</td>
<td>$(i,j)$-edge's capacity</td>
</tr>
<tr>
<td>$b_i$</td>
<td>Vertex $i$'s supply or demand indicator</td>
</tr>
<tr>
<td>$x_{ij},y_{ij}$</td>
<td>The flow values along $(i,j)$-edge</td>
</tr>
<tr>
<td>$\delta$</td>
<td>A distance function</td>
</tr>
<tr>
<td>$\mathcal{O}$</td>
<td>The Bachmann-Landau notation</td>
</tr>
<tr>
<td>$\mathcal{P},\mathcal{NP}$</td>
<td>Complexity classes</td>
</tr>
<tr>
<td>$\mathcal{P},\mathcal{Q},\mathcal{R}$</td>
<td>Sets of problems</td>
</tr>
<tr>
<td>$x_i,y_i,z_i$</td>
<td>Boolean variables</td>
</tr>
<tr>
<td>$v_i$</td>
<td>Literals</td>
</tr>
<tr>
<td>$A_i,B_i$</td>
<td>Subgraphs for the reduction</td>
</tr>
</tbody>
</table>
Chapter 1

Background

In this chapter, we introduce some basic background of this research, have a brief review of what studies have been done prior to our work, and present a practical motivation for choosing such topic.

1.1 Introduction

This research is in the area of Analytics, also known as Operations Research (OR), and more specifically in optimization models and methods. Optimization, encompassing theory, numerical algorithms and applications, is ubiquitous to many sciences and engineering, and thus occupies one of the pillars of analytics – the scientific process of transforming data into insight for making better decisions. Specifically, we investigate novel means to improve the resilience of a certain supply chain by rigorous mathematical modelling.

Catastrophic changes in operational and natural environments present a huge risk and uncertainty in supply chains. Obviously, if one could easily predict catastrophes, these events would not cause an issue in most cases. By building redundancy into the system, as is done with infrastructure used in power transmission, the system becomes more robust and less likely to fail as a result of unforeseen circumstances. A natural question arises: how can we add the necessary level of redundancy for the least possible cost? Our work is largely
motivated by a practical question — optimal transmission powerline routing — as further described in Section 1.3.

A common abstraction of the transmission powerline routing problem is to model the system as a graph. In turn, many utility metrics are reduced to graph properties, such as the shortest path between two vertices, etc. Mathematically, a graph $G$ is an ordered pair $(V, E)$ consisting of a set $V$ of vertices (or nodes) and a set $E$ of edges connecting the vertices. The two vertices of an edge are said to be incident with the edge, and vice versa. A path is presented by a sequence of edges connecting the origin and destination. A graph can be directed — when the edges are allowed to be traversed in the specified direction only — or undirected. To help differentiate between different edges in terms of their desirability, it is common to assign an individual cost (or weight) to each edge. In the latter case, the graph is called weighted.

As the theoretical underpinning of our study, we investigate the problem of shortest paths, and precisely, a variant of this problem using multiple paths subject to geometric constraints. This variant is poorly understood. The shortest path problem forms a fundamental abstract setting for many applied problems where the best, i.e., the cheapest, the fastest, the most sustainable, etc., route or set of routes needs to be found. These types of problems are also paramount to many other sub-branches of optimization and discrete mathematics. For example, as noted in [2] pp.93, “shortest path problems lie at the heart of network flows.” The original shortest path problem ([1] pp.150) is stated as follows.

**Problem 1.1.** Given a weighted graph $G$ with two specified vertices $s$ and $t$, find a minimum-weight $(s,t)$-path in $G$.

For the sake of consistency with Graph Theory and Computer Science terminology, we call the initial vertex $s$ the source and the terminal vertex $t$ the sink. In Figure 1.1, we illustrate the notion of the shortest path where the shortest path with respect to the cost metric is depicted in red.

The original shortest path problem is one of the biggest successes of classical OR stem-
Figure 1.1: An example of a shortest path.

...ming from the original work of Dijkstra in the late 1950s [3]. To date, the shortest path problem has found many applications in network design, GPS routing, etc. Despite the original shortest path is now classified as an “easy” computational problem, some of its innocent-looking variants present incomparable challenges.

In this thesis, we study one such variant which we refer to as the Disjoint Shortest 2-path Problem (DS2P), in which two disjoint paths need to be determined instead of one, and in addition, the paths have to conform to some minimal distance constraints. By distance constraints, we mean that the two paths have to be at least a certain distance apart from each other. The distance constraints obviously have to be relaxed or even dropped altogether for the edges near the source and the sink nodes, as otherwise the model ceases to be meaningful and becomes infeasible. We will formalize this problem further in Chapter 2.

Our main contribution is to propose a novel method to account for geometric constraints such as distance constraints that, under certain further assumptions, guarantees optimal 2-path recovery in reasonable time. The method is based on a geometric embedding of a 2-path problem in a higher dimensional graph, followed by the shortest path computation, say, using Dijkstra’s algorithm or its variant. In the embedding model, hard geometric constraints are converted to the graph’s connectivity map with the associated cost structure. The computational run-time requirements to recover an optimal set of geometrically distinct 2 paths grow very moderately as compared to all the other known alternatives, resulting in the efficient numerical procedure. Along the way, we also give a precise and more conventional 0-1 integer programming formulation of the 2-path problem, and demonstrate the
comparative advantages of our embedding approach. The method can be generalized to accommodate distinct origin-destination pairs, more complicated geometric constraints, and a larger number of distinct paths desired. We also propose a graph-theoretic model for DS2P and prove its \( \mathcal{NP} \)-hardness in general.

This thesis is organized as follows: in Chapter 2, we formulate the problem into two models, namely the graph-based model and the 0-1 programming model; in Chapter 3, we discuss some basic properties of the exact models, including the complexity of the graph-based model and the linear programming relaxation of the 0-1 programming model; in Chapter 4, we introduce our novel approximation scheme and compare its performance to the 0-1 programming model.

1.2 Literature review

In many applications a necessity arises to determine a set of alternative paths between some source-sink pair. For example, a multi-path concept was applied to network robustness in the context of transit design in [17]-[20], where several groups of authors studied the survivability of low cost communication networks. Some of the proposed models fall within the domain of so-called bilevel programming, while others pursue a graph-theoretic approach.

Transmission line routing and planning has long drawn attention as an application of the shortest path problem as well; see for example [10]-[15]. Much work in the literature focuses on expansion planning, where the routes are well documented by transmission owners or regulators [16]. Classical engineering reliability aspects of power systems are already well covered in the literature, for example see [21] and [22]. The optimal placement of the new transmission lines corresponds to the topological design of a very specialized unorthodox “supply chain”, where multiple power lines serve to increase the system’s resilience against catastrophic failures.

Besides Dijkstra’s algorithm [3], many other efficient and specialized algorithms exist to
determine a single shortest (or least-cost) path between any two nodes in the graph, such as the Bellman–Ford algorithm [4]-[5], the Floyd–Warshall algorithm [6]-[7], etc. Generally, these algorithms are very efficient computationally and allow one to find the shortest path for very large graphs with relative ease, typically requiring on the order of $|\mathcal{E}| + |\mathcal{V}| \ln |\mathcal{V}|$ number of arithmetic operations to complete, where $|\mathcal{E}|$ and $|\mathcal{V}|$ denote the number of edges and vertices in the graph respectively.

One extension of the classical shortest path problem is the so-called $K$ shortest path routing problem. Here, the aim is to find not only one single shortest path but also the other $(K - 1)$ paths in non-decreasing order of cost. Due to its importance, the $K$ shortest path problem is well-studied. For brevity we give only two entry points to this field of research, namely [8] and [9] where polynomial-time algorithms were given. There are also versions of algorithms determining the existence of $\kappa$ disjoint paths between $\kappa$ distinct pairs of vertices $(s_1, t_1), (s_2, t_2) \ldots (s_\kappa, t_\kappa)$ in a given graph $G$. This variant of the $\kappa$ disjoint paths problem has also been extensively studied as well, and again, for brevity, we only cite [23] here where polynomial algorithms are proposed.

Unlike the classical single shortest path problem, or the above $K$ shortest paths routing problem or the $\kappa$ disjoint paths problem, little is known about the 2-path variant with geometric constraints. Although the latter problem also has many potential applications, to our knowledge, no general-purpose and efficient method exists for DS2P where the user can specify such a geometric constraint. Indeed, mostly heuristic methods lacking rigorous analysis have been proposed. For instance, when two paths are desired, usually a manual search is performed on a set of $K$-disjoint paths, instead of systematically recovering the cheapest pair of paths. In a typical powerline application, the size of the graphs representing possible network topology is very large (millions of nodes alone). As a consequence of using unproven heuristics, one may end up using sub-optimal solutions. In turn, this could result in large socio-economic losses.
1.3 Motivational example

The Northeast region of Alberta, Canada, is an area with the highest growth in electricity demand across the province. Oil sand industries in the Fort McMurray area constitute a large portion of the current and future electricity demand. On the other hand, a large amount of hydro energy is available in the Slave River basin, which is located at the northern border of the Alberta province. The problem is how to transmit electricity to the Fort McMurray area. The northern region has a very sparse population and most of the area is covered with forests and ice, and there is almost no human development in the area. The hydro-generated power has to be transmitted via power lines to substations in the vicinity of oil sand facilities in Fort McMurray. Through this, it is possible to bring green energy to energy intensive oil sand operations and contribute to greenhouse gas reduction plans.

Figure 1.2: Cost color-wash map.

There are many geographical, environmental, economic, human-footprint, and other considerations in the region. To quantify these, the regional map may be discretized with a
certain resolution and assigned a representative “weight” for every cell in the grid, corresponding to a power-line transitioning through the specific cell. The resulting color-wash map is presented in Figure 1.2, with red and darker colors corresponding to more expensive regions, in terms of the anticipated construction cost. The detailed map is divided into 311-by-244 cells of $1 \times 1$ km cell size resolution. Two extremely dark areas correspond to the forbidden zones, where large water bodies are located and the cost of transmission line construction may as well be assumed to be infinite.

![Figure 1.3: Hypothetical Northeastern Alberta power-line topology.](image)

The objective is to find the optimum route or corridor for the powerlines in Northeastern Alberta. A hypothetical Northeastern Alberta power-line topology example is presented in Figure 1.3, with a star depicting the generator facility, and a circle depicting the receiver station. When designing a power transmission line between the generating facility and the consumer node, a company may be legally required to build not just one but two redundant power lines, distanced from one another by some minimal safety margin, so that if one of the lines goes down due to an accident, there is still a high probability of power being delivered
to the destination via the second back-up line.

Thus, our objective is to find two separate corridors for the transmission lines going from the power plant to the substation. Because of the specific reliability standards, these two lines must be separated by a distance of at least 50 km, or equivalently, 50-unit cells apart. The latter problem may be abstracted to solving an instance of DS2P — finding the shortest 2-path on a graph subject to a geometric distance constraint with vertices representing cell centers and edges representing the cost of transitioning from any one cell to another.

Our research fits within the general Analytics platform and is highly applicable. Similarly to the power-line design problem, a constrained 2-path problem naturally arises in road construction, high-capacity fiber-optic cable network design for telecom, and GPS navigation. An efficient solution to the 2-path problem would benefit all of these areas, which are currently driven by heuristic sub-optimal methods. The problem may also have immediate applications to the so-called minimal energy protein folding in computational chemistry and biophysics that naturally leads to yet another completely different set of applications.
Chapter 2

Problem formulations

In this chapter, we propose two formulations of DS2P, each following a completely different viewpoint. The first formulation approaches the problem from a graph-theoretic viewpoint. The second formulation is based on so-called mixed-integer programming, or more specifically, 0-1 integer programming. Both formulations may be regarded as general models in a sense that the distance constraints can take more exotic and abstract forms. However, since this work is motivated by a practical problem, we pay the most of attention to a specific graph structure — the weighted uniform grid — with a Euclidean or counting metric measuring the distance. The first formulation is used to establish a hardness result for the corresponding variant of DS2P. On the other hand, the second model is closely related to the famous min-cost flow formulation of the shortest path problem which is defined and discussed in Section 2.2, and thus, arguably, may be used as a more convenient baseline to perform computational comparisons of the performance of our novel scheme against more classical approaches.

2.1 Graph-based model

For simplicity, from now on we assume we are dealing with a finite directed graph $G$ with vertices indexed by $V = \{1, 2, 3, ..., \eta\}$. Let $(i, j)$ represent the arc going from vertex $i$ to $j$, 

with associated weight $c_{ij}$. Denote the source by $s$ and the sink by $t$ as it is in Problem 1.1.

Recall that a path in a directed graph from the vertex $u$ to $v$ is a collection of consecutive vertices leading from $u$ to $v$, where some other vertices may be visited along the way. Correspondingly, the ordered sequence of the vertices $\langle u, v \rangle = \{u, i_1, i_2, ..., v\}$ visited along the path between $u$ and $v$ is used for path encoding. Note that between any two vertices in a graph there may be no path at all, the unique path, or a set of alternative paths, depending on the graph’s configuration.

We say that the $\langle u, v \rangle$-path $\{u, i_1, i_2, ..., i_{\kappa-1}, v\}$ is of length $\kappa$, in accordance with the number of vertices visited. That is, the corresponding path vertex sequence has $\kappa + 1$ entries. For example, a single edge corresponds to a path of length of 1. The path $\langle u, v \rangle$ has *cumulative weight* (or cost) $c_{\langle u, v \rangle}$, computed as $\sum_{n=0}^{\kappa-1} c_{i_n i_{n+1}}$ with $i_0 \equiv u$ and $i_\kappa \equiv v$.

**Definition 2.1.** Vertex $u$ is $\Delta$-reachable from $v$ if there exists a $\langle u, v \rangle$-path of length $\Delta$ or less.

**Definition 2.2.** A neighborhood of a vertex $v \in G$ denoted by $N_v$ is a subset of $V$ such that $u \in N_v$ iff $u$ is $\Delta$-reachable from $v$ for some $\Delta$.

We introduce neighborhoods here because the distance constraint would obviously have to be relaxed or even dropped altogether for the edges near the source and the sink, otherwise the model ceases to be meaningful and becomes infeasible. Now we are in position to state our first and rather more abstract variant of DS2P.

**Problem 2.3.** Given a directed graph $G$, a source $s \in N_s$, a sink $t \in N_t$ with two neighborhoods $N_s$ and $N_t \subset V$, and a fixed distance threshold $\Delta$, find two paths from $s$ to $t$, call them $\langle s, t \rangle^{(a)}$ and $\langle s, t \rangle^{(b)}$, such that

(i) the two paths are at least $\Delta$-distance apart, meaning that

for all $i_a \in \langle s, t \rangle^{(a)} \setminus (N_s \cup N_t)$

and all $i_b \in \langle s, t \rangle^{(b)} \setminus (N_s \cup N_t)$

the vertex $i_a$ is not $(\Delta - 1)$-reachable from $i_b$ and vice versa;
(ii) the total cost of the two paths, \(c_{(s,t)^{(a)}} + c_{(s,t)^{(b)}}\) is minimized amongst all possible path alternatives satisfying (i).

Figure 2.1: Illustration of Problem 2.3

See Figure 2.1 as an illustration. In particular, if \(\Delta = 1\), \(\mathcal{N}_s = \{s\}\) and \(\mathcal{N}_t = \{t\}\), Problem 2.3 asks to find 2 vertex-disjoint (except at \(s\) and \(t\)) shortest paths from \(s\) to \(t\). An example for a \(K_{3,3}\) directed graph is shown in Figure 2.2. Since \(K_{3,3}\) is non-planar ([1], pp.245) meaning it cannot be drawn in a plane so that its edges intersect only at their ends ([1], pp.243), we cannot use Euclidean metric to characterize the distance constraints. However, the \(\Delta\)-distance we have defined as the metric fits general graphs — planar or non-planar.

Figure 2.2: A \(K_{3,3}\) directed graph

Note that there are three paths from Node 1 to Node 6 in Figure 2.2, namely \(\{1,5,3,6\}\), \(\{1,4,2,6\}\) and \(\{1,4,2,5,3,6\}\). If we want to solve Problem 2.2 with \(\Delta = 1\), \(\mathcal{N}_s = \{s\}\) and
\( \mathcal{N}_t = \{t\} \), then we can only pick \( \{1,5,3,6\} \) and \( \{1,4,2,6\} \), since they have no common vertex except the source Node 1 and the sink Node 6.

The scenario when \( \Delta = 2, \mathcal{N}_s = \{s\} \) and \( \mathcal{N}_t = \{t\} \), is very important, and its significance will be seen in Section 3.1 where complexity is discussed.

2.1.1 Real-world scenario

For practical reasons and simplicity, we assume that the cost graph corresponds to a uniform square \( K \)-by-\( K \) grid. We also assume that the grid is oriented as a diamond directed graph, where the source corresponds to the top vertex, and the sink corresponds to the very bottom vertex as in Figure 2.3 with edges oriented so that they point down. Note that this \( K \)-by-\( K \) grid is a graph with \( K^2 \) vertices. We only use diagonal edges, with \( 2K(K - 1) \) of those present. The goal is to find the optimal pair of two paths which are at least \( \Delta \) units apart.

![Figure 2.3: 3-by-3 diamond directed graph](image)

Although initially it may seem that such a graph choice is very restrictive, we argue that in fact that it is not quite the case. That is, from the practical (planar) perspective, the diamond grid is a flexible model. For instance, we note that our diamond-shaped grid can easily accommodate vertical edges as well. This requires a small modification to the original grid, which we illustrate with the example shown in Figure 2.4, where a vertical edge of cost 4 from Node 1 to Node 4 is given. We refine the graph in the following way: set the middle
points of edges (1,2), (1,3), (2,4), (3,4) and (1,4) to be new vertices, namely 1’, 2’, 3’, 4’ and 5’, and divide the corresponding edge weights into two. Connect 1’, 2’, 3’ and 4’ to 5’ respectively. In order to assign weights to those newly-added edges, we introduce imaginary vertical edges (1,5’) and (5’,4). Since 5’ is in the middle of edge (1,4), we set the costs of imaginary edges (1,5’) and (5’,4) to be 2. Then assign the costs of edge (1’,5’) and (2’,5’) to be 0.5 and 1 respectively so that the total costs of paths {1,1’,5’} and {1,2’,5’} are both 2 which is the same as the cost of the imaginary edges (1,5’). In that way, the path {1,1’,5’} or {1,2’,5’} functions equivalently to the imaginary vertical edge (1,5’). Similarly, {1,1’,5’,3’,4}, {1,2’,5’,3’,4}, {1,1’,5’,4’,4} or {1,2’,5’,4’,4} functions as equivalent to the vertical edge (1,4). Hence, by adjusting the graph resolution, we can accommodate for the vertical edges by having diagonal edges exclusively.

Figure 2.4: Graph with vertical edges

In practice, the actual points of interest may not be exactly on the grid, and the edges between these points may be curved as shown in Figure 2.5. The example looks very circuitous, however, it is still very easy to rectify — the vertices can be thought of as coming from the uniform grid while the edge weights are correctly assigned.

One of the main reasons to restrict ourselves to the $K$-by-$K$ diamond grid is the case of implementation of our proposed scheme. In short, this abstraction allows for streamlined memory management when passing to the graph, and this outweighs other concerns such as the grid seeming to be too restrictive, etc.
2.2 0-1 integer programming model

As we have mentioned, the bulk of the 0-1 integer programming model relies on the so-called min-cost flow model for the shortest path. We introduce the min-cost flow model first, and then we propose the 0-1 integer programming formulation for both exact and real-world scenario.

2.2.1 Min-cost-flow-type model

A flow over an edge is a real number assigned to the edge. Intuitively, for example, it is the amount of goods carried on the road, or the amount of water flushing through the tube. The min-cost flow problem is the central object of study in almost every book on network flows. Besides its theoretical importance, problems which are modelled by min-cost flow constantly arise in industries including manufacturing, communication, transportation and so on.

The min-cost flow problem is typically defined over a capacitated network, that is, in addition to the edges cost $c_{ij}$, there is a capacity $u_{ij}$ for each edge $(i, j)$, signifying that the flow along $(i, j)$-edge cannot exceed $u_{ij}$. Now, the min-cost flow model is given as follows.([2] pp.296)

Problem 2.4. Suppose we associate a number $b_i$ to each vertex $i \in \mathcal{V} = \{1, 2, ..., \eta \}$, which indicates whether the vertex is a supply or demand node, depending on whether $b_i \geq 0$ or
The min-cost flow problem corresponds to solving the following linear programming problem, where \( x_{ij} \) represents the flow along \((i,j)\)-edge, and is given in the following way:

\[
\min_x \sum_{(i,j) \in \mathcal{E}} c_{ij} x_{ij}
\]

s.t. \( \sum_{j : (i,j) \in \mathcal{E}} x_{ij} - \sum_{j : (j,i) \in \mathcal{E}} x_{ji} = b_i, \ \forall i \in \mathcal{V}, \)

\[
0 \leq x_{ij} \leq u_{ij}, \ \forall (i,j) \in \mathcal{E}.
\]

In its generic form, the min-cost flow problem allows for fractional optimal flow values. However, under certain assumptions it yields integer flows only. Specifically, when all the capacities \( u_{ij} \) and \( b_i \) are integers, all integer optimal flow values \( x_{ij} \) may be recovered.

The model may look deceptively restrictive at first glance. In fact, the min-cost flow formulation is very flexible by just modifying \( b_i \) and \( u_{ij} \). By making clever choices for \( b_{ij} \) and \( u_{ij} \), we can model many situations including the shortest path. We will illustrate this flexibility by showing the formulation for the original shortest path Problem 1.1 as a min-cost flow formulation.

First, observe that the shortest path problem may be formulated as the following integer programming problem (with binary variables),

**Problem 2.5.**

\[
\min_x \sum_{(i,j) \in \mathcal{E}} c_{ij} x_{ij}
\]

s.t. \( \sum_{j : (i,j) \in \mathcal{E}} (x_{ij} - x_{ji}) = \begin{cases} 1, & i = s \\ -1, & i = t \\ 0, & \text{otherwise} \end{cases} \)

\[
x_{ij} \in \{0,1\}, \ \forall (i,j) \in \mathcal{E}.
\]

Here, the binary variables \( x_{ij} \) are used to indicate whether \((i,j)\)-edge is used in the construction of the shortest path — set \( x_{ij} = 1 \) — or not — set \( x_{ij} = 0 \). The set of affine
constraint guarantees the flow conservation. The source $s$ has one unit of flow out and the sink $t$ has one unit of flow in. All the other vertices always have zero-net flow since only one unit of flow is allowed to go in one out. Note that if we change the “1” and “−1” in the affine constraint to “$\kappa$” and “−$\kappa$” respectively, we will generalize the model to get precisely $\kappa$ edge-disjoint shortest paths from $s$ to $t$, where the total cost of all the paths is minimized.

Finally, we note that due to the above integrality property, the binary variables may be replaced with continuous capacitated ones, $0 \leq x_{ij} \leq 1$, $\forall (i,j) \in E$, thus giving us an instance of min-cost flow.

Based on the min-cost flow model, we can easily formulate the DS2P problem by adding one more flow corresponding to the $y$-variable and the required distance constraint $\delta((i,j),(i',j')) \leq \Delta$ where $\delta$ is a function to measure the distance between $(i,j)$-edge and $(i',j')$-edge and $\Delta$ is the required minimum distance.

**Problem 2.6.**

\[
\begin{align*}
\min_{x,y} \sum_{(i,j) \in E} c_{ij}(x_{ij} + y_{ij}) \\
\text{s.t.} \quad & \sum_{j:(s,j) \in E} x_{sj} - \sum_{j:(j,s) \in E} x_{js} = 1, \quad \sum_{j:(s,j) \in E} y_{sj} - \sum_{j:(j,s) \in E} y_{js} = 1, \\
& \sum_{j:(t,j) \in E} (x_{tj} - \sum_{j:(j,t) \in E} x_{jt}) = -1, \quad \sum_{j:(t,j) \in E} y_{tj} - \sum_{j:(j,t) \in E} y_{jt} = -1, \\
& \sum_{j:(i,j) \in E} x_{ij} - \sum_{j:(j,i) \in E} x_{ji} = 0, \quad \sum_{j:(i,j) \in E} y_{ij} - \sum_{j:(j,i) \in E} y_{ji} = 0, \quad \forall i \neq s, t \\
& x_{ij} + y_{i'j'} \leq 1, \text{ for all } i, j, i', j' \in V \setminus (N_s \cup N_t) \text{ such that } \delta((i,j),(i',j')) \leq \Delta \\
& x_{ij}, y_{ij} \in \{0,1\}, \quad \forall i, j \in V
\end{align*}
\]

Similiar to Problem 2.5, in Problem 2.6, the first three constraints are set to maintain the flow conservation, while the last of the inequalities are used to guarantee the required distance. Due to the same reason that the distance constraint is not applicable near the source and the sink, we relax the constraints in the properly chosen neighborhoods of the
source and the sink. By restricting the cumulative flow $x_{ij} + y_{i'j'}$ to 1 we clearly permit only one path within the close proximity to the other. Note that we intentionally do not specify the distance function $\delta$ to keep the model as flexible as possible. The distance function can be modified to adjust to different desired outcomes and may or may not depend on the specific metric embedding of the network. For example, the model can be used to solve the graph-based version of DS2P presented in the previous subsection properly by defining “$\delta((i,j),(i',j')) \leq \Delta$” as “at least one of the vertex-pairs $\{i,j\}, \{i',j\}$, $\{i,j'\}$ or $\{i',j'\}$ is not $\Delta$-distance apart”.

### 2.2.2 Real-world scenario (continued)

Unlike the rather abstract notion of distance illustrated above, in practical applications it may use a Euclidean-like distance function to measure the edges’ proximity. Specifically, in the diamond directed graph in Section 2.1.1, we can apply the Euclidean distance or Manhattan distance for the distance measure function $\delta^\epsilon$ for any pair of edges $(i,j)$ and $(i',j')$ in $G$:

$$\delta^\epsilon((i,j),(i',j')) = \min\{\text{dist}(i,i'), \text{dist}(i,j'), \text{dist}(j,i'), \text{dist}(j,j')\},$$

where “dist” is the Euclidean distance or Manhattan distance function measuring the distance between any two nodes.

For notational convenience, we add a spurious sink-to-source edge with the flow value of 1 as shown in Figure 2.6. Then the exact 0-1 programming formulation of the constrained 2-path problem may be stated as follows.
Figure 2.6: 3-by-3 diamond directed graph with a spurious sink-to-source edge

Problem 2.7.

\[
\min_{x,y} \sum_{(i,j) \in E} c_{ij}(x_{ij} + y_{ij})
\]
\[
\text{s.t. } \sum_{i : (i,j) \in E} x_{ij} = \sum_{k : (j,k) \in E} x_{jk}, \quad \sum_{i : (i,j) \in E} y_{ij} = \sum_{k : (j,k) \in E} y_{jk}, \quad \forall j \in \mathcal{V}
\]
\[
x_{K^2,1} = y_{K^2,1} = 1
\]
\[
x_{ij} + y_{i'j'} \leq 1, \text{ for all } i, j, i', j' \in \mathcal{V} \setminus (\mathcal{N}_s \cup \mathcal{N}_t) \text{ such that } \delta^e((i, j), (i', j')) \leq \Delta
\]
\[
x_{ij}, y_{ij} \in \{0, 1\}, \quad \forall i, j \in \mathcal{V}
\]

Note that the last model corresponds to solving an instance of 0-1 programming which in general may become notoriously difficult. A natural question arises: can we simply relax the binary constraints into simple linear inequalities to greatly speed up the solution process? We discuss the model’s properties in Subsection 3.2, where we present a fractional-flow example.
Chapter 3

Basic properties of DS2P models

In this chapter, we discuss some basic complexity properties of DS2P, mainly the complexity of solving a DS2P problem for general graphs, and whether the binary constraints can be relaxed in order to get a much easier linear programming problem to solve.

3.1 Graph-based model’s complexity

Complexity Theory is a branch of theoretical Computer Science and Mathematics. It provides a classification of problems according to how much time it will take for algorithms to solve. A time complexity function for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size ([27] pp.6). In other words, the time complexity function indicates the algorithm’s performance in the worst case. There are several classes indicating how “hard” a problem is, such as \( \mathcal{P}, \mathcal{NP}, \mathcal{NP}\)-complete and \( \mathcal{NP}\)-hard. In this subsection, we will briefly review some fundamental concepts in Complexity Theory, and then discuss the hardness of the graph-based DS2P problem.
3.1.1 Basic concepts review

First recall the Bachmann-Landau notation $O$, a.k.a the big O notation. Following that, we define the notion of polynomial-time algorithm.

**Definition 3.1.** ([27] pp.6) A function $f(n) : \mathbb{N} \to \mathbb{R}$ is $O(g(n))$ whenever there exists a constant $c$ such that $|f(n)| \leq c \cdot |g(n)|$ for all values of $n \geq 0$.

**Definition 3.2.** ([27] pp.6) If $n$ denotes the input length, a polynomial-time algorithm is defined to be one whose time complexity function is $O(p(n))$ for some polynomial function $p$.

Instead of going too far into Computer Science, we attempt to define different complexity classes rather informally in a more understandable way without mentioning the Turing machine.

**Definition 3.3.** $\mathcal{P}$, which stands for “polynomial time”, is the class of problems that are solvable in polynomial time, i.e., there exists a polynomial-time algorithm to solve it.

**Definition 3.4.** $\mathcal{NP}$ which stands for “nondeterministic polynomial time” is the class of problems that is verifiable in polynomial time, i.e., given any result there exists a polynomial-time algorithm to verify if it is true.

Is there any relation between $\mathcal{P}$ and $\mathcal{NP}$? Of course, directly from these two definitions, we have $\mathcal{P} \subseteq \mathcal{NP}$. Is $\mathcal{P} = \mathcal{NP}$? In fact, the $\mathcal{P}$ versus $\mathcal{NP}$ problem is one of the seven Millennium Prize Problems stated by the Clay Mathematics Institute in 2000 and is still unknown.

A natural question is: given two sets of problems $P$ and $Q$, how can we compare their difficulties? The indispensable technique is reduction. Roughly speaking, reduction is an algorithm for transforming one problem into another problem and shows that the latter one is at least as difficult as the former.
**Definition 3.5.** Let $P$ and $Q$ be two sets of problems. We say that $P$ is reducible to $Q$ or $Q$ is the reduction from $P$, denoted by $P \leq Q$, if there exists a function $f$ such that

- $f$ is polynomial-time computable, that is, given a problem $x \in P$, $f(x)$ can be computed in polynomial time;
- $x \in P$ if and only if $f(x) \in Q$.

Intuitively, solving $Q$ is not easier than solving $P$, and an algorithm designed for solving $Q$ can also be used for solving $P$. On one hand, $Q$ can be regarded as $P$'s general case, on the other hand $P$ can be regarded as a special case of $Q$. For example, let $P$ be the set of solving linear equations $kx + t = 0$ and let $Q$ be the class of problems of solving quadratic equations $ax^2 + bx + c = 0$, where all the parameters are real. Let $f : P \rightarrow Q$ be the inclusion mapping. That is,

$$P = \{ \text{solving } kx + t = 0 | k, t \in \mathbb{R} \}$$
$$Q = \{ \text{solving } ax^2 + bx + c = 0 | a, b, c \in \mathbb{R} \}$$

and $f : "\text{solving } kx + t = 0" \rightarrow "\text{solving } 0x^2 + kx + t = 0"$.

Given any $x = "\text{solving } kx + t = 0" \in \mathcal{R}$, $f(x) = "\text{solving } 0x^2 + kx + t = 0"$ is computed by simply adding one more term "0x^2", and the “adding” step can be done in polynomial time, thus, $f$ is polynomial-time computable. And because of the way we have constructed $f$, it is obvious that $x \in P$ if and only if $f(x) \in Q$. Therefore, we conclude that $P \leq Q$.

From its definition, we also have two more immediate observations:

- if $P \leq Q$ and $Q \in \mathcal{P}$, then $P \in \mathcal{P}$ (conversely, if $P \leq Q$, and $P \notin \mathcal{P}$ then $Q \notin \mathcal{P}$);
- if $P \leq Q$ and $Q \leq R$, then also $P \leq R$.

With Definition 3.4, we can define two more classes of problems — $\mathcal{NP}$-complete and $\mathcal{NP}$-hard.

**Definition 3.6.** $\mathcal{NP}$-complete is the class of problems satisfying

- any problem in $\mathcal{NP}$-complete is also in $\mathcal{NP}$;
any problem in \( \mathcal{NP} \) is reducible to a problem in \( \mathcal{NP} \)-complete in polynomial time.

**Definition 3.7.** \( \mathcal{NP} \)-hard is the class of problems that are at least as hard as the hardest problems in \( \mathcal{NP} \).

Basically, it means the problems in \( \mathcal{NP} \)-hard are very, very hard to solve. It is obvious that \( \mathcal{NP} \)-complete problems are in \( \mathcal{NP} \)-hard, since all the problems in \( \mathcal{NP} \) are no harder than them. The relation of the four classes is illustrated in Figure 3.1.

There are several problems have been proven to be \( \mathcal{NP} \)-complete. For example, the linear programming (LP) problem, long known to be \( \mathcal{NP} \) and thought not to be \( \mathcal{P} \), was shown to be \( \mathcal{P} \) by L. Khachian presenting his ellipsoid algorithm [28] for LP in 1979. Another important example of \( \mathcal{NP} \)-complete problem is the 3SAT problem which makes a key contribution to Section 3.1.2. We give the following definitions first.

**Definition 3.8.** A Boolean expression consists of binary variables (0 means false while 1 means true), three operations — AND (a.k.a conjuction, denoted by \( \land \)), OR (a.k.a disjunction, denoted by \( \lor \)) and NOT (a.k.a negation, denoted by a bar above the variable) — and parentheses.

Figure 3.1: Relation of \( \mathcal{P} \), \( \mathcal{NP} \), \( \mathcal{NP} \)-complete and \( \mathcal{NP} \)-hard
Definition 3.9. The Boolean satisfiability problem, abbreviated as SAT, is the problem of determining if an assignment exists to satisfy a given Boolean expression. When there exists such one, the assignment is called a true assignment and the given Boolean expression is satisfiable.

For example, \( x_1 = 1, x_2 = 0 \) and \( x_3 = 1 \) is a true assignment for the Boolean expression \( x_1 \land (\bar{x}_1 \lor x_2) \land (x_1 \lor x_2 \lor \bar{x}_3) \), and consequently this Boolean expression is satisfiable.

To be consistent with terms in Computer Science, we introduce the following jargons: a literal is either a variable, then called positive literal, or the negation of a variable, then called negative literal; a clause is a disjunction of literals, or a single literal. The 3-SAT problem is a type of Boolean satisfiability problem with specific structure, that is each clause contains exactly three literals. For example, finding a true assignment for \((\bar{x}_1 \lor x_2 \lor \bar{x}_3) \land (x_1 \lor \bar{x}_2 \lor x_3) \land (x_1 \lor x_2 \lor \bar{x}_3)\) is a 3SAT problem. We will need the following result in Section 3.1.2.

Theorem 3.10. ([27] pp.48) The 3-SAT problem is \( \mathcal{NP} \)-complete.

3.1.2 Hardness classification

Interestingly, the behavior of the DS2P varies drastically with the desired distance between the paths. Specifically, when \( \Delta = 1 \), the problem is very easy, while for \( \Delta \geq 2 \) it becomes very hard. We proceed to illustrate this phenomenon in detail.

As mentioned in Subsection 2.1, \( \Delta = 1 \) leads to the vertex-disjoint shortest paths problem. Polynomial algorithms are given for the vertex-disjoint shortest paths problem in undirected graphs with positive weights in [23]-[25]. Note that a digraph can be modified to fit the min-cost flow model [26]. We illustrate this modification procedure with the example illustrated in Figure 3.2 where the solid lines have cost of 1 and the dashed lines have cost of 10.

Suppose we want to find two vertex-disjoint shortest paths from Node 1 to Node 4 in Figure 3.2. If we simply implement the min-cost flow model, we will get \( \{1,2,3,4\} \) and
Figure 3.2: Digraph modification for $\Delta = 1$.

$\{1,3,5,4\}$ whose total cost is zero. However, these two paths have a common vertex: Node 3. The modification guarantees that every node is visited only once, since the edge $(3, 3')$ can be only used once. As we have mentioned before, the min-cost flow model has property of having integer flow (more details in Section 3.2) and since the LP problem is in $\mathcal{P}$, we conclude that the problem with $\Delta = 1$ is in $\mathcal{P}$ as well.

When considering DS2P and its possible relatives, it is interesting to mention that the following problem (stated in [23]) is $\mathcal{NP}$-complete.

**Problem 3.11.** Given a graph $G$ with positive edge weights and two pairs of vertices $(s_1, t_1)$, $(s_2, t_2)$, find whether there exist two disjoint (edge-disjoint or vertex-disjoint) paths $P_1$ from $s_1$ to $t_1$ and $P_2$ from $s_2$ to $t_2$ such that $P_1$ is a shortest path.

Indeed, inspired by the reduction technique of Problem 3.11 in [23], we have the following theorem:

**Theorem 3.12.** Let $\mathcal{N}_s = \{s\}$ and $\mathcal{N}_t = \{t\}$.

If $\Delta = 1$, then DS2P is in $\mathcal{P}$

If $\Delta \geq 2$, then DS2P is in $\mathcal{NP}$-hard.

We claim that the DS2P problem with $\Delta = 2$ is also $\mathcal{NP}$-hard by showing the 3SAT problem is reducible to it.

**Proof.** Denote the number of clauses and the number of literals in the expression by $m$ and $n$ respectively. For each clause $(x_i \lor y_i \lor z_i)$ we construct a subgraph $A_i \ (1 \leq i \leq m)$, and for each pair of literals $v_j$ and $\bar{v}_j$ we construct a subgraph $B_j, 1 \leq j \leq n$ as shown in Figure 3.3.
$A_i$ has nine edges and the three middle ones represent the corresponding literals $x_i, y_i$ and $z_i$. $B_j$ has two sides – the left side consists of the edges of $\bar{v}_j$ and the right side consists of the edges of $v_j$. Denote the times that each literal $v_j$ appears in all $A_i$'s by $|v_j|$, and then set the length of both sides of $B_j$ be $l_j = 2 \max\{|v_j|, |\bar{v}_j|\} + 1$, so that both sides have $l_j$ edges of $v_j$ and $\bar{v}_j$ respectively. Note that this construction process is a polynomial-time transformation.

![Figure 3.3: 3SAT reduction subgraphs](image)

Connect each $A_i$ ($1 \leq i \leq m$) sequentially by coinciding one's last vertex with the next one's first vertex, and repeat the same process for $B_j$'s. Then connect each pair of vertices which are incident with the edge of the literal $v_k$ ($1 \leq k \leq n$) in $A_i$'s to the ones in $B_j$'s incident with the edge of $\bar{v}_k$ with dashed lines respectively and sequentially. We use solid lines to connect the source $s$ to the initial points $s_1$ and $s_2$ of subgraphs $A$ and $B$ respectively and connect the sink $t$ to the ending points $t_1$ and $t_2$ as well, and we use dashed lines to connect $s_1$ to $t_1$ and $s_2$ to $t_2$. Set the weight of all the solid lines to be 1 and the weight of all the dashed lines to be $3m + \sum_{j} l_j$. For example, Figure 3.4 is constructed for $(x_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_1 \lor x_4) \land (\bar{x}_2 \lor x_3 \lor \bar{x}_4)$ and Figure 3.5 is constructed for $(x_1 \lor x_2 \lor x_2) \land (x_1 \lor \bar{x}_2 \lor \bar{x}_2) \land (\bar{x}_1 \lor x_2 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2 \lor \bar{x}_2)$.

We claim that finding two disjoint shortest paths subject to the distance constraint $\Delta = 2$ from $s$ to $t$ in Figure 3.4 is equivalent to solving the 3SAT problem $(x_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_1 \lor x_4) \land (\bar{x}_2 \lor x_3 \lor \bar{x}_4)$. Note that Figure 3.4 has the following properties.
1. The DS2P is always feasible: there are always two paths satisfying all the constraints, that is \( \{ s, s_1, t_1, t \} \) and \( \{ s, s_2, t_2, t \} \) whose total cost is \( 2(3m + \sum_j l_j + 2) \);

2. Every variable admits a unique assignment: if the first path \( P_1 \) goes through the edge \( x_1 \) in \( A_1 \), the second path \( P_2 \) cannot go through the edge \( \bar{x}_1 \) in \( B_1 \) due to the distance constraint \( \Delta = 2 \);

3. The DS2P optimum determines the satisfiability. To be convenient, we denote that one path goes through subgraphs \( A_i \)'s (or \( B_j \)'s) by \( A \) (or \( B \)) and otherwise by \( \bar{A} \) (or \( \bar{B} \)); and by \( A \cup \bar{B} \), we mean that, under the distance constraint, the two situations \( A \) and \( \bar{B} \) co-occur,
that is one of the two path goes through all $A_i$’s while the other does not go through all $B_j$’s. There are four scenarios:

(a) $A \cup B$, and then the least cost is $\text{cost}_a = 3m + \sum_j l_j + 4$;

(b) $A \cup \bar{B}$, and then the least cost is $\text{cost}_b = 6m + \sum_j l_j + 4$;

(c) $\bar{A} \cup B$, and then the least cost is $\text{cost}_c = 3m + 2\sum_j l_j + 4$;

(d) $\bar{A} \cup \bar{B}$, and then the least cost is $\text{cost}_d = 2(3m + \sum_j l_j + 2)$.

It is straightforward that

$$\text{cost}_a < \min\{\text{cost}_b, \text{cost}_c\} \leq \max\{\text{cost}_b, \text{cost}_c\} < \text{cost}_d$$

and Scenario (a) implies the corresponding 3SAT problem has a true assignment.

In the example shown in Figure 3.4, there exist such two disjoint shortest path:

$$P_1: \{s, s_1, x_1(A_1), x_4(A_2), \bar{x}_2(A_3), t_1, t\}$$

and $P_2: \{s, s_2, x_1(B_1), \bar{x}_2(B_2), x_3(B_3), x_4(B_4), t_2, t\}$.

If we set all the literals in $P_2$ to be true, that is

$$x_1 = 1, \ \bar{x}_2 = 1, \ x_3 = 1, \ \text{and} \ x_4 = 1,$$

then $(x_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_1 \lor x_4) \land (\bar{x}_2 \lor x_3 \lor \bar{x}_4) = 1$ and we solve the 3SAT problem.

In the example shown in Figure 3.5, there do not exist such two paths with the total cost $3m + \sum_j l_j + 4$ that satisfy the distance constraint. And it is not a coincidence that the corresponding 3SAT problem $(x_1 \lor x_2 \lor x) \land (x_1 \lor x_2 \lor x) \land (\bar{x}_1 \lor x_2 \lor x) \land (\bar{x}_1 \lor x_2 \lor x)$ has no true assignment, meaning it is always false.

Considering these two example together, we can see that indeed the outcome of solving a DS2P problem implies the satisfiability of its corresponding 3SAT expression. If the outcome
Figure 3.5: The graph constructed for \((x_1 \lor x_2 \lor x_2) \land (x_1 \lor \bar{x}_2 \lor \bar{x}_2) \land (\bar{x}_1 \lor x_2 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2 \lor \bar{x}_2)\) shows the cost equals \(3m + \sum_j l_j + 4\), then it has a true assignment which can be told by the path from \(s_2\) to \(t_2\); otherwise, the 3SAT expression is always false.

Graphs constructed for scenarios when \(\Delta \geq 3\) can be simply regard as subdivisions of the graph of scenario when \(\Delta = 2\). Also note that this reduction applies regardless of the graph being directed or undirected, hence, for both types of graphs the DS2P problem with \(\Delta \geq 2\) is \(\mathcal{NP}\)-hard.
3.2 LP relaxation of the 0-1 programming model

Recall in Section 2, we have presented the 0-1 programming formulation which is based on the min-cost flow problem. The biggest advantage which makes the original Min-Cost Flow formulation so classic is that the binary requirements $x_{ij} \in \{0, 1\}$ can be relaxed to $x_{ij} \in [0, 1]$ so that the potentially very hard problem is converted to be a polynomial-time solvable problem. The latter feature of the min-cost flow formulation of the classical shortest path problem is commonly referred to as the Integrality Theorem ([2] pp.381).

**Theorem 3.13.** If all edge capacities and supplies/demands of nodes are integers, the min-cost flow problem always has integer optimal min-cost flow values.

Since all the parameters in Problem 2.5 are integer, the integrality of the min-cost flow problem is guaranteed. Thus, a very natural conjecture is whether we can do the same relaxation on the binary variables in Problem 2.6 or 2.7. For example, consider the following linear programming relaxation of Problem 2.7:

**Problem 3.14.**

$$\min_{x,y} \sum_{(i,j) \in E} c_{ij} (x_{ij} + y_{ij})$$

s.t. \(\sum_i x_{ij} = \sum_k x_{jk}, \sum_i y_{ij} = \sum_k y_{jk}, \forall j \in V\)

\(x_{K^2,1} = y_{K^2,1} = 1\)

\(x_{ij} + y_{i'j'} \leq 1, \text{ for all } i, j, i', j' \in V \setminus (N_s \cup N_t) \text{ such that } \delta^e((i, j), (i', j')) \leq \Delta\)

\(x_{ij}, y_{ij} \in [0, 1], \forall i, j \in V\)

First, we should note that the cardinality of the distance constraints set grows asymptotically proportional to $16K^2$. Similarly, prescribing the distance of at most $\Delta$ units between the paths would result in the number of inequality constraints growing proportional to $\Delta^2$, in turn leading to a quick, albeit polynomial, growth of the problem dimensions in $K$ and
\( \Delta \). As a consequence, even when Problem 3.12 is considered, for relatively modest practical values of \( K \) and \( \Delta \), e.g. \( K = 300 \) and \( \Delta = 2 \), solving such problems may already pose a significant computational challenge.

Another essential downside of using Problem 3.14 instead of Problem 2.7 is that the LP relaxation may result in fractional flows.

![Fractional-flow example](image)

*Figure 3.6: Fractional-flow example*

Suppose we want to solve a DS2P problem for the uniform grid graph shown in Figure 3.6 where Node 1 is the source, Node 16 is the sink, all the solid lines have cost 1 and dashed lines have cost \( \infty \). Clearly, we want the paths to go along the solid lines as otherwise the total cost of the paths would be infinite.

If we further require the Euclidean distance to be strictly bigger than \( \sqrt{2} \) with the exception of vertices immediately adjacent to the source and sink, then there do not exist two such paths with finite cost, since the Euclidean distance between Node 2 and Node 3 is exactly \( \sqrt{2} \). However, it can be easily verified that the LP relaxation may yield 4 fractional flows.
(paths) as a feasible solution —

\[
\begin{align*}
    x_{1,2} &= x_{2,4} = x_{4,8} = x_{8,11} = x_{11,14} = x_{14,16} = 0.5, \\
    y_{1,2} &= y_{2,4} = y_{4,8} = y_{8,11} = y_{11,14} = y_{14,16} = 0.5, \\
    x_{1,3} &= x_{3,6} = x_{6,9} = x_{9,13} = x_{13,15} = x_{15,16} = 0.5, \\
    \text{and } y_{1,3} &= y_{3,6} = y_{6,9} = y_{9,13} = y_{13,15} = y_{15,16} = 0.5,
\end{align*}
\]

which implies two of them overlap with each other along the same edges with ratios all 0.5.

We would like to re-emphasize that solving Problem 2.7 may represent a huge computational challenge, as the number of binary variables alone grows proportional to \(K^2\), and thus increases rapidly with the size of the grid \(K\). More importantly, the number of the constraints required to enforce that the two paths are far apart also grows super-linearly in the dimension, making the problem practically intractable for large \(K\) where \(K\) approaches thousands.

We conjecture that the DS2P posed on a regular grid graph also exhibits \(\mathcal{NP}\)-hardness, which, in very loose terms means the problem is as hard as the hardest decision problems known to date.
Chapter 4

A novel approximation scheme

4.1 3D Embedding

4.1.1 Assumptions yielding optimality

As an alternative to the 0-1 programming formulation, we propose a much more efficient approximation approach to finding the two disjoint paths, which under mild additional assumptions results in an exact solution. We state these assumptions first. Strictly speaking, they are not necessary to solve the approximation scheme proposed, however, the assumptions will help us to streamline the scheme’s description that follows.

(I) \textit{(Horizontal separability)} Let the vertical spacing between the vertices of the graph be such that the distance between the paths can be judged solely by the horizontal distance between the visited nodes.

(II) \textit{(Synchronicity)} Let the two paths be \textit{synchronous} in vertical direction, that is, both paths descend one vertical step at a time.
4.1.2 Procedure

Roughly speaking, the key idea is to establish a correspondence between a 2D problem and a 3D problem. The correspondence, which is technically almost one-to-one, is defined in the sense that any pair of two paths in the 2D graph corresponds to a path in the 3D graph, and vice versa, given a 3D path between the 3D source \((0,0,0)\) and sink \((0,0,K)\), the corresponding two paths can be found by simple orthogonal projections onto 2D, as illustrated in Figure 4.1. For example, the edge \(\{(i_1,i_2,j),(i'_1,i'_2,j+1)\}\) in 3D projects to \(\{(i_1,j),(i'_1,j+1)\}\) and \(\{(i_2,j),(i'_2,j+1)\}\) in 2D. It follows that we can build a 3D graph, based on the 2D graphs, with “unqualified” edges removed due to the distance constraint, compute the single shortest path in 3D and project it onto the 2D planes. Indeed, Assumptions (I) and (II) guarantee the one-to-one property, if we differentiate the order of the pair of paths, that is, the pair of \(\{(i_1,j),(i'_1,j+1)\}\) and \(\{(i_2,j),(i'_2,j+1)\}\) is different from the pair of \(\{(i_2,j),(i'_2,j+1)\}\) and \(\{(i_1,j+1),(i'_1,j+1)\}\). Thus, the two assumptions yield optimality.

Figure 4.1: Demonstration of 3D embedding
With the above in mind, we elaborate the following geometric embedding procedure. We take a 4-by-4 2D map in Figure 4.2 as an example demonstrating the procedure graphically. For convenience, we call the horizontal line containing (0,0) Layer 0 and so forth until Layer $2K - 2$ which is Layer 6 in this example.

![Figure 4.2: 2D map](image)

First, replicate the graph twice, and assign the graph realizations to two orthogonal planes, say $XZ$ and $YZ$. For each of the graph realizations, we align the corresponding $Z$-axis with the line connecting the source (0,0) and sink (6,0), while the $X$-axis and $Y$-axis represent the orthogonal directions to the $Z$-axis.

Our next step is to merge the two 2D graphs into a single 3D graph as shown in Figure 4.3. First, we align both source and sink of the two graphs. Next, out of the two planar edge-cost graphs we will produce a single 3D graph in the $XYZ$ space. Let the vertices of the 3D graph be indexed by triples of coordinates as $(x, y, z)$ whose 2D projections are $(x, z)$ in the $XZ$ plain and $(y, z)$ in the $YZ$ plain. Note that $z$ coordinates of both paths must agree at this pair of points. For example, the 3D vertex $(-1,1,1)$ encodes a pair of vertices $(-1,1)$
in the $XZ$ plain and $(1,1)$ in the $YZ$ plain, both of which have $z = 1$. Similarly, the 3D edge $(0,0,0),(-1,1,1)$ encodes a pair of edges $(0,0),(-1,1)$ and $(0,0),(1,1)$ in 2D, and consequently, the cumulative 3D edge cost is the sum of costs of the two corresponding projections in 2D, that is, $c_{(0,0,0),(-1,1,1)} = c_{(0,0),(-1,1)} + c_{(0,0),(1,1)}$.

Figure 4.3: 3D embedding

Most importantly, under Assumptions (I) and (II) the previously “hard” geometric distance constraint can be encoded into the 3D graph by simply omitting the 3D edges that result in two projected edges that are too close. Thus we proceed with omitting such edges or, alternatively, assigning the infinite cost to the 3D edges to be omitted.

As a result, in order to find a pair of disjoint shortest paths, it suffices to determine a single shortest path on the resulting 3D graph. The latter task may be easily accomplished, say, by Dijkstra’s algorithm as shown in Figure 4.4, at a very modest computational cost.
Figure 4.4: Single shortest path in 3D

Then we project the single 3D shortest path in Figure 4.4 onto the 2D plane respectively as shown in Figure 4.5.

Figure 4.5: Projections on 2D maps
In the end, combine them together and get two paths in one 2D plane as shown in Figure 4.6.

![Figure 4.6: Two paths in 2D](image)

Assumptions (I) and (II), albeit restrictive, may be easily verified in some situations. Simply put, (I) depends on the coarseness of the cost grid, while (II) relies on the optimal paths not containing loops or “stagnating” one with respect to the other. We note that even when assumptions (I) and (II) are not met, the embedding scheme may be modified to produce an approximate solution to the DS2P problem.

### 4.2 Preliminary models’ performance and discussion

Next, we compare the performance of the two models, namely, we consider how the 3D embedding scheme performs relative to the 0-1 based formulations in Section 2.2.

Both models are benchmarked using Matlab environment. Since the 0-1 IP model sizes grow very rapidly with the size of the grid $K$ and the prescribed minimal distance, for simplicity, we consider only the time required to solve the LP relaxation of the problem. In turn, it is reasonable to expect that the 0-1 IP solution times cannot be any lower, if not far greater. For further equity of comparison, in both cases of solving the relaxed 0-1 IP and the embedding models we use open source solvers. Namely, in the former case of
solving an LP relaxation we use the well-reputed SDPT3 conic solver, while in the latter of solving a shortest path problem, we use the Dijkstra’s algorithm script available from the Mathworks/Matlab Bioinformatics Toolbox.

In Table 4.1, we report run-times for the two approaches, as they scale with problem dimension $K$. The computations were completed using a (modest) Surface laptop with 8G RAM and a dual-core Intel CPU running up to 3.3GHz. The problem instances were randomly generated, meaning that the cost-edge matrix was sampled uniformly, and we report average run-time values over a sample of 10 problems for each dimensionality. A minimal distance requirement of 2 units between the paths was imposed.

<table>
<thead>
<tr>
<th>$K$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1 IP/LP</td>
<td>0.9880</td>
<td>1.4950</td>
<td>5.5994</td>
<td>43.1325</td>
<td>422.8889</td>
<td>4635.7253</td>
</tr>
<tr>
<td>Building 3D adjacency matrix</td>
<td>0.0026</td>
<td>0.0063</td>
<td>0.0419</td>
<td>0.5907</td>
<td>5.2739</td>
<td>51.0341</td>
</tr>
<tr>
<td>Dijkstra’s algorithm</td>
<td>0.0007</td>
<td>0.0013</td>
<td>0.0084</td>
<td>0.1540</td>
<td>1.3583</td>
<td>31.1758</td>
</tr>
</tbody>
</table>

Table 4.1: Average run-time in seconds for the two schemes; for the 0-1 IP scheme we only solve its root LP relaxation.

Note that, as expected, the computational demands imposed by solving even a relaxed version of the 0-1 IP formulation are notably greater, as compared to the 3D embedding scheme. For more realistic problem dimensions, with $K$ going into hundreds, in our experiments solving even a relaxed version of the 0-1 IP formulation quickly became infeasible. Again, we want to emphasize that the main difficulty in solving such LPs is attributed to the geometric constraints, where specifying the constraints, even in the relaxed form, is an exhausting task. In contrast, the cost of solving the 3D embedding model remained nearly negligible as compared to solving the LP, making it reasonable to assume that the method may be adapted to a real-time analysis of the network topology for realistic grid sizes.

Indeed, in our experiments, solving the motivating problem instance discussed in Section 1, based on the factual data and operating on a 180-by-180 grid, took less than 3 minutes; see Figure 4.7 for an example of the paths pair found. In this specific application, rather than a single destination, two distinct destination nodes—power sub-stations preceding the
end consumption node in the power system—were identified. Thus, two separate paths, each leading to an individual substation, needed to be determined. The presented embedding scheme could be easily modified to accommodate such an extension: if the two sinks in 2D have the same height, say \((t_1, K)\) and \((t_2, K)\), then set the sink to be \((t_1, t_2, K)\) in 3D; if the two sinks have different heights, then we need rotate the graph first so as to make the heights the same — of course, rotation will lead to loss of precision of the original data.

For better visual comparison, we depict the optimal pair of the distance-disjoint paths in sub-figure (a), and the two shortest paths to each of the individual terminal nodes, computed individually with the distance constraint omitted, in sub-figure (b), Figure 4.7. Clearly, the two solutions are distinct from one another. For the individual shortest-path solutions, the two paths coincide until both enter the close proximity of the two destination nodes, at which point they finally split up.

![Figure 4.7: Optimal disjoint 2-path pair (a) against 2 individual shortest paths (b).](image)

We would like to re-emphasize that no specialized shortest path solver was used, and
the computations were performed on a modest workstation using free-ware Matlab scripts. In principle, if finer grid granularity and speedier computations are desired, one can switch to a more powerful solver. It is important to note that one of the limiting factors in the implementation of the embedding scheme may appear to be the memory requirement, as the 3D graph grows fairly quickly with $K$. However, we note that, in fact, the 3D graph structure, in principle, can be easily computed “on the flight” during the computations of the shortest path algorithm, eliminating this bottleneck as further described in the next subsection.

4.3 Advanced implementation for large-scaled graph

When the Dijkstra’s algorithm in Matlab is implemented, its input is the adjacency matrix. If the regular grid is $K$ by $K$, then the 3D graph has $(2 \sum_{n=1}^{K} n^2 - K^2)$ vertices, and then the number of entries of the adjacency matrix becomes $(2 \sum_{n=1}^{K} n^2 - K^2)^2$. When the grid is large, its 3D adjacency matrix may overflow the memory, and this drives us to find another way to compute the cost of the 3D graph.

We want to take the full advantage of the regular grid which has a specific structure. Firstly, note that the two vertices in 3D, $(i_1, i_2, j)$ and $(i_2, i_1, j)$ have identical projections, $(i_1, j)$ and $(i_2, j)$, to the 2D graph. Thus, it is sufficient to consider each horizontal cross section as a triangle instead of a square.

Secondly, according to the optimality assumptions in Section 4.1, the vertex-costs of the current layer are only based on the former layer. This implies we can compute the costs layer by layer. To illustrate our last comment, consider the following. The 3D graph starts from Layer 0 which has only one vertex $(0,0,0)$. See Figure 4.8 for example where the pentagons represents vertices on Layer 2 and circles represents vertices on Layer 3. From $(2,-2,2)$, the path may go to any of $(3,-3,1), (3,-3,3), (3,-1,1)$ or $(3,-3,3)$ — we set the directions corresponding to each to them to be SW, NW, SE and NE which are short for southwest,
northwest, southeast and northeast respectively. Suppose we have already computed all the costs of paths of from \((0,0,0)\) to Layer 2, in order to compute the cost of paths from \((0,0,0)\) to Vertex \((3,-1,1)\), the information needed is just the edge costs from \((0,0,0)\) to \((2,-2,0), (2,-2,2), (2,0,0)\) and \((2,0,2)\). We compute all the costs from the four different directions respectively and pick the minimum to be the path cost from \((0,0,0)\) to \((3,-1,-1)\). Then proceed to the next layer, i.e. Layer 4, after all the costs of getting to Layer 3 are computed in the former way. Note that only two consecutive layers are needed, thus when computing Layer 4, we can delete all the cost information before Layer 2 that is stored. However, we do need to keep the direction information of all the layers in order to recover the shortest path by backtracking from the bottom (sink) to the top (source).

<table>
<thead>
<tr>
<th>(K)</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building 3D adjacency matrix</td>
<td>5.2739</td>
<td>25.2071</td>
<td>51.0341</td>
<td>151.7399</td>
<td>270.6072</td>
</tr>
<tr>
<td>Dijkstra’s algorithm</td>
<td>1.3583</td>
<td>10.2498</td>
<td>31.1758</td>
<td>79.6350</td>
<td>284.0086</td>
</tr>
<tr>
<td>Layered approach</td>
<td>3.0506</td>
<td>5.8450</td>
<td>8.8574</td>
<td>19.5514</td>
<td>29.2346</td>
</tr>
</tbody>
</table>

Table 4.2: Average run-time in seconds for 3D embedding: naïve vs layered approach
In Table 4.2, we compare the run-times of the original 3D embedding scheme with Dijkstra’s algorithm and the advanced layer-by-layer scheme, when dealing with larger-scale graphs. All the computations were completed using the same Surface laptop as the one for Table 4.1. In order to fit Matlab, we compute the costs vector-wise, that is, for example, we regard (3,-3,-3), (3,-3,-1), (3,-3,1) and (3,-3,3) as a whole vector and compute its cost at the same time. The gap becomes bigger when the scale grows.

To further speed up the computations, implementation of algorithms in parallel scheme can be applied. Parallel computing is a type of computation following the basic idea that large problems can often be divided into smaller ones which can be solved at the same time. This computation has been employed in recent years, mainly in tackling large-scale problems and high-performance computing. In particular, general-purpose computing on graphics processing units (GP-GPU) is a very powerful technique for parallel computing, which typically handles computation only for computer graphics. GP-GPU is the use of a graphics processing unit (GPU) to perform computation in applications traditionally handled by the central processing unit (CPU). [29]. It may have greater potential in high-performance computing, thus further implementation will be done on GP-GPU.
Chapter 5

Conclusion and future work

We investigated the question of determining an optimal transmission line layout in the context of resilient power supply. Specifically, the problem of how to determine a configuration for the set of two (one main and one redundant) power lines, connecting the generator and the receiver, was addressed, where the additional minimal distance between the paths requirement was imposed, and the total cost of construction was to be minimized. We formulated the problems into a graph-based model and a 0-1 integer programming model, and classified the hardness for the former model as well as discussed the LP relaxation for the latter model. Along with the two exact formulations, we presented a far more efficient numerical scheme, based on a novel 3D graph embedding, which under some further mild assumptions leads to the exact solution. The embedding scheme may be extended to accommodate more than two paths, several distinct destinations, etc. In addition to the motivating application, the proposed DS2P framework befits many other potential applications, such as road design and so on.

An interesting and important question remains if the embedding scheme can be used to improve the otherwise dramatic run-times required by the exact 0-1 programming model, for example, when the additional assumptions leading to the scheme’s optimality cannot be met. Also, the complexity classification for the problem on the regular grid is not complete.
yet. Finally, we would like to port our layered implementation of the new scheme onto heterogeneous computing platform to exploit the use of powerful GP-GPU to improve the numerical performance considerably. These items are the subject of the ongoing work.
Bibliography


Appendix A

MATLAB codes

A.1 DiamondAdj.m

```matlab
function [Hash,CE,M] = DiamondAdj(K,L_edges,R_edges,ss,dist)
  % From the ss-th layer, 2 paths are dist units apart from each other.
  dist = ceil(dist/2)*2;

  % How many rows of Hash, i.e. how many vertices in 3D?
  % How many rows of CE, i.e. how many edges in 3D?
  Vsum = 0;
  for j = 1:K-1
    Vsum = Vsum + sum(1:j);
  end
  Vsum = Vsum*2 + sum(1:K); Hash = zeros(Vsum,3);

  % Unqualified vertices remain 0 since the graphshortestpath function take 0
  % as infinity. And be careful that 0 in the inputs L_edges & R_edges should
  % be replaced with NaN. Those unqualified vertices needn’t recorded in CE,
  % so we deduct their number.
```

49
Esum = 0;

for j = 1:ss
    Esum = Esum + sum(1:2*j);
end

for j = ss+1:K-1
    Esum = Esum + 4*sum(1:j-dist/2);
end

Esum = Esum*2; CE = zeros(Esum,3);

%% Hash is easy:

cntr = 0;

for j = 0:K-1
    for i1 = -j:2:j
        for i2 = i1:2:j
            cntr = cntr+1;
            Hash(cntr,:) = [j,i1,i2];
        end
    end
end

for j = K:2*(K-1)
    layer_num = 2*(K-1)-j;
    for i1 = -layer_num:2:layer_num
        for i2 = i1:2:layer_num
            cntr = cntr+1;
            Hash(cntr,:) = [j,i1,i2];
        end
    end
end
% CE is kind of easy...

% This part has 2 different vertices - the one on the hypotenuse and the % one inside.

for j = 0:ss-1
    for i1 = -j:2:j
        % Say j = 3, first locate (3,-3,-3) which has 3 adjacent vertices:
        cntr = cntr + 3;
        vertex = label_vertex(j,i1,i1);
        vertexLL = label_vertex(j+1,i1-1,i1-1); % LL: (4,-4,-4)
        vertexLR = label_vertex(j+1,i1-1,i1+1); % LR: (4,-4,-2)
        vertexRR = label_vertex(j+1,i1+1,i1+1); % RR: (4,-2,-2)
        edge = label_edge(i1,j);
        CE(cntr-2,:) = [vertex,vertexLL,L_edges(edge)*2];
        CE(cntr-1,:) = [vertex,vertexLR,L_edges(edge)+R_edges(edge)];
        CE(cntr,:) = [vertex,vertexRR,R_edges(edge)*2];
        for i2 = i1+2:2:j % Start from (3,-3,1) with 4 adjacent vertices:
            cntr = cntr + 4;
            vertex = label_vertex(j,i1,i2);
            vertexLL = label_vertex(j+1,i1-1,i2-1); % LL: (4,-4,-2)
            vertexLR = label_vertex(j+1,i1-1,i2+1); % LR: (4,-4, 0)
            vertexRL = label_vertex(j+1,i1+1,i2-1); % RL: (4,-2,-2)
            vertexRR = label_vertex(j+1,i1+1,i2+1); % RR: (4,-2, 0)
            edge1 = label_edge(i1,j); edge2 = label_edge(i2,j);
            CE(cntr-3,:) = [vertex,vertexLL,L_edges(edge1)+L_edges(edge2)];
            CE(cntr-2,:) = [vertex,vertexLR,L_edges(edge1)+R_edges(edge2)];
            CE(cntr-1,:) = [vertex,vertexRL,R_edges(edge1)+L_edges(edge2)];
            CE(cntr,:) = [vertex,vertexRR,R_edges(edge1)+R_edges(edge2)];
    end
end
end
end

for j = ss:K-2
  for i1 = -j:2:j-dist
    for i2 = i1+dist:2:j
      cntr = cntr + 4;
      vertex = label_vertex(j,i1,i2);
      vertexLL = label_vertex(j+1,i1-1,i2-1); % LL
      vertexLR = label_vertex(j+1,i1-1,i2+1); % LR
      vertexRL = label_vertex(j+1,i1+1,i2-1); % RL
      vertexRR = label_vertex(j+1,i1+1,i2+1); % RR
      edge1 = label_edge(i1,j); edge2 = label_edge(i2,j);
      CE(cntr-3,:) = [vertex,vertexLL,L_edges(edge1)+L_edges(edge2)];
      CE(cntr-2,:) = [vertex,vertexLR,L_edges(edge1)+R_edges(edge2)];
      CE(cntr-1,:) = [vertex,vertexRL,R_edges(edge1)+L_edges(edge2)];
      CE(cntr,:,:) = [vertex,vertexRR,R_edges(edge1)+R_edges(edge2)];
    end
  end
end
end

for j = K:2*K-2-ss
  jc = 2*K-2-j; % j complement.
  for i1 = -jc:2:jc-dist
    for i2 = i1+dist:2:jc
      cntr = cntr + 4;
      vertex = label_vertex(j,i1,i2,K);
      vertexLL = label_vertex(j-1,i1+1,i2+1,K); % LL
      vertexLR = label_vertex(j-1,i1+1,i2-1,K); % LR

52
vertexRL = label_vertex(j-1,i1-1,i2+1,K);  % RL
vertexRR = label_vertex(j-1,i1-1,i2-1,K);  % RR
edge1 = label_edge(i1,j,K);  edge2 = label_edge(i2,j,K);
CE(cntr-3,:) = [vertexLL,vertex,L_edges(edge1)+L_edges(edge2)];
CE(cntr-2,:) = [vertexLR,vertex,L_edges(edge1)+R_edges(edge2)];
CE(cntr-1,:) = [vertexRL,vertex,R_edges(edge1)+L_edges(edge2)];
CE(cntr,:) = [vertexRR,vertex,R_edges(edge1)+R_edges(edge2)];
end
end
end

for j = 2*K-1-ss:2*K-2
    jc = 2*K-2-j;  % j complement.
    for i1 = -jc:2:jc
        cntr = cntr + 3;
        vertex = label_vertex(j,i1,i1,K);
        vertexLL = label_vertex(j-1,i1+1,i1+1,K);  % LL
        vertexLR = label_vertex(j-1,i1+1,i1-1,K);  % LR
        vertexRR = label_vertex(j-1,i1-1,i1-1,K);  % RR
        edge = label_edge(i1,j,K);
        CE(cntr-2,:) = [vertexLL,vertex,L_edges(edge)*2];
        CE(cntr-1,:) = [vertexLR,vertex,L_edges(edge)+R_edges(edge)];
        CE(cntr,:) = [vertexRR,vertex,R_edges(edge)*2];
        for i2 = i1+2:2:jc  % Start form (3,-3,1) with 4 adjacent vertices:
            cntr = cntr + 4;
            vertex = label_vertex(j,i1,i2,K);
            vertexLL = label_vertex(j-1,i1+1,i2+1,K);  % LL
            vertexLR = label_vertex(j-1,i1+1,i2-1,K);  % LR
            vertexRL = label_vertex(j-1,i1-1,i2+1,K);  % RL
        end
    end
end
vertexRR = label_vertex(j-1,i1-1,i2-1,K); % RR
edge1 = label_edge(i1,j,K); edge2 = label_edge(i2,j,K);
CE(cntr-3,:) = [vertexLL,vertex,L_edges(edge1)+L_edges(edge2)];
CE(cntr-2,:) = [vertexLR,vertex,L_edges(edge1)+R_edges(edge2)];
CE(cntr-1,:) = [vertexRL,vertex,R_edges(edge1)+L_edges(edge2)];
CE(cntr ,:) = [vertexRR,vertex,R_edges(edge1)+R_edges(edge2)];
end
end
end

M = sparse(CE(:,1),CE(:,2),CE(:,3),cntr,cntr);
end

function vertex = label_vertex(j,i1,i2,K)
% Given a coordinate, what is the cumulative label?
if nargin == 3
    vertex = (i1+j)/2*(j+1)+(i2+j)/2+1-sum(1:(i1+j)/2)+... 
            (j*(j+1)*(2*j+1)/6+(j+1)*j/2)/2;
else
    % Upper = K^3/6+K^2/2+K/3;
    Upper = K*(K+2)*(K+1)/6;
    Total = 2*Upper - sum(1:K);
    jc = 2*K - 2 - j;
    vertex = Total - label_vertex(jc,jc,jc) + ... 
              (i1+jc)/2*(jc+1)+(i2+jc)/2+1-sum(1:(i1+jc)/2);
end
end

54
function edge = label_edge(i,j,K)
% Given a 2D coordinate, how to find the 2 edges strating from this vertex?
% v1=(0,0)
% L1/ \R1
% v2=(-1,1) v3=(1,1)
% L2/ \R2 L3/ \R3
% v4=(-2,2) v5=(0,2) v6=(2,2)
% L4/ \R4 L5/ \R5 L6/ \R6
% v7=(-3,3) v8=(-1,3) v9=(1,3) v10=(3,3) useless layer...
% R7\ /L7 R8\ /L8 R9\ /L9
% v11=(-2,4) v12=(0,4) v13=(2,4)
% R10\ /L10 R11\ /L11
% v14=(-1,5) v15=(1,5)
% R12\ /L12
% v16=(0,6)
if nargin == 2
    edge = (i+j)/2+1+sum(1:j);
else
    edge = K-K*K+(i+j)/2+(4*K-3-j)*j/2;
end
end

A.2 path3D.m

function [cost_parent] = path3D(K,L_edges,R_edges,ss,dist,dest)
% This is a new vector version.
%_statuses

Prelude
\% K*K 2D graph -> L_edges & R_edges (num of each: (K-1)*K)
\%
\% \(/
\% \//\  
\% \\\
\% \\\\ -> \(4/5/6/\ + \ \(4\backslash 5\backslash 6\)
\% \\\
\% \\\
\% \\\
\% \\\
\% \\\
\% From the ss-th layer where ss is required to be bigger than dist or 3?
\% 2 paths start separating dist units.
\%
\% The 2D graph’s vertices are labelled in the following natural way:
\% 
\% \(0,0\)
\% \(4, 1\) \((-1,1)\) \((1,1)\)
\% \(8, 5, 2\) \((-2,2)\) \((0,2)\) \((2,2)\)
\% \(12, 9, 6, 3\) \((-3,3)\) \((-1,3)\) \((1,3)\) \((3,3)\)
\% \(13, 10, 7\) \((-2,4)\) \((0,4)\) \((2,4)\)
\% \(14, 11\) \((-1,5)\) \((1,5)\)
\% \(15\) \((0,6)\)
\% Our destination doesn’t have to be exactly the last one No.15
\% Given a destination dest, (we can also use the rotation matrix)
destlevelup = dest - floor(dest/K)*(K-1); \% Presumaly >= K.
desthorizon = dest - floor(dest/K)*(K+1);
dist = ceil(dist/2)*2;

\% Right now, 2 paths are guaranteed to be ">dist" unites apart from each
\% other, but if you want ">=dist", one can simply let dist = dist-2;
dist = dist - 2;
Again, it means from (ss+1)th to (K-1)th, 2 paths are at least dist units
apart, e.g. (3,-3) and (3,-1) are 2 units apart.

We may assume the destlevelup >= K-1 since we can always chop the graph.

dest = 8 or dest = 5

0

4 1 0

8 5 2 4 1 0

12 9 6 3 -> 8 5 2 or 4 1

13 10 7 5

14 11

15

Be careful: L_edges & R_edges are 2 arrays (i.e. vectors)!

R

^ / SW=LL=1

NW | NE/ NW=LR=2

| / SE=RL=3

L-- -->R NE=RR=4

/ |

/SE SW | SE

/ L

Note that we only need a half!

The first 4 layers: 0th -> 1st -> 2nd -> 3rd

v4-v7-v9-v10

v3-v5-v6 | | | / 

v2-v3 | | / v3-v6-v8

v1 -> | / -> v2-v4 -> | | /
% v1   | /   v2-v5
% v1   | / 
% v1

% Planform e.g. the 2nd & 3rd layers
% (3,-3, 3)--(3,-1, 3)--(3, 1, 3)--(3, 3, 3)
% \   /   / \   / 
% (2,-2, 2) | (2, 0, 2) | (2, 2, 2)
% / \   \ /   \ / 
% 1 (3,-3, 1)--(3,-1, 1)--(3, 1, 1)
% \   /   /   / 
% | (2,-2, 0) | (2, 2, 0)
% \ /   \ /   \ / 
% 1 (3,-3,-1)--(3,-1,-1)
% \   / 
% | (2,-2,-2)
% \ / 
% 1 (3,-3,-3)

%---------------------------------------------------------------
% Upper & Unconstrained
%---------------------------------------------------------------

% First part is unconstrained, i.e. no distant constraints.
% 0th layer is just (j,i1,i2)=(0,0,0); 1st layer is
% v2-v3  (1,-1, 1)-(1, 1, 1)
% | /   ->   |
% v1  (1,-1,-1)
% start from the 2nd layer; vertex costs come from L_edge(1) & R_edges(1);
cost_parent = zeros(3,1); dir = zeros(3,1);
cost_parent(1) = L_edges(1) + L_edges(1);  % (1,-1,-1), No.1
cost_parent(2) = L_edges(1) + R_edges(1); % (1,-1, 1), No.2
cost_parent(3) = R_edges(1) + R_edges(1); % (1, 1, 1), No.3
% It is obvious -1 represents left while 1 represents right.
% For the 1st layer, directions are fixed
dir(1) = 1; % LL or SW
dir(2) = 2; % LR or NW (RL or SE corresponds to 3)
dir(3) = 4; % RR or NE

% display(dir); display(cost_parent);
Vidx = fopen('pathIDX.txt', 'w+');
fprintf(Vidx,'%d ',0);
fprintf(Vidx,'%d ',dir);
fclose(Vidx);

for j = 2:ss
% For the j-th layer, the number of vertices is sum(1:j+1).
Vnum = (j+1)*(j+2)/2; cost_kid = zeros(Vnum,1); dir = ones(Vnum,1);

% For the 1st column vector, whichever direction to choose, LL or LR,
% the projection on first plane is always the same left edge. Say j=3:
% The parent of (3,-3,-3) is (2,-2,-2), the same left-edged projection
% is L_edges(label_temp), and we need compare the common part [v2 v3]
% cost_kid(1:j)=[v1 v2 v3] which correponds to dir = 1 or LL
% cost_temp = [v2 v3 v4] which correpsonds to dir = 2 or LR
% Since they are corners, v1 is always dir=1 and v4 is always dir=2.
label_edge = label_3(1-j,j-1); % Locate the same left-edged projection.
cost_kid(1:j) = cost_parent(1:j) + L_edges(label_edge:label_edge+j-1);
cost_temp = cost_parent(1:j) + R_edges(label_edge:label_edge+j-1);
% Compare cost_kid(2:j) with cost_temp(1:j-1)
for i = 2:j
    if cost_kid(i) > cost_temp(i-1);
        cost_kid(i) = cost_temp(i-1);
        dir(i) = 2;
    end
end
cost_kid(j+1) = cost_temp(j); dir(j+1) = 2; % The NW corner vertex.
% Add the same left-edged projection:
cost_kid(1:j+1) = cost_kid(1:j+1) + L_edges(label_edge);

for col = 2:j % Vector computation for each column.
    % The 2nd vector (3,-1,3)-(3,-1,1)-(3,-1,-1) has length of 3.
    length = j+2-col; range = length-2;
    i_temp = 2*(col-1)-j; % Compute the "-1" of (3,-1,-1).
    label_kid = label_1(j,i_temp,i_temp); % Locate (3,-1,-1).

    % First compare the pair of LL & LR, since they have the common
    % left-edged projection which is to be added in the end.
    label_parent = label_1(j-1,i_temp+1,i_temp+1); % Locate (2,0,0).
    % Locate the edge from (i2,j)=(0,2) which is the different part.
    label_edge = label_3(i_temp+1,j-1);
    cost_kid(label_kid:label_kid+range) = ...
        cost_parent(label_parent:label_parent+range)+ ...
        L_edges(label_edge:label_edge+range); % LL
    cost_temp = cost_parent(label_parent:label_parent+range)+ ...
        R_edges(label_edge:label_edge+range); % LR
    for i = 1:range
        label_temp = label_kid+i;
        if cost_kid(label_temp) > cost_temp(i);
\begin{verbatim}
cost_kid(label_temp) = cost_temp(i);

dir(label_temp) = 2;
end
end

% The NW corner vertex:
cost_kid(label_kid+length-1) = cost_temp(length-1);
dir(label_kid+length-1) = 2;
% Add the same left-edged projection:
cost_kid(label_kid:label_kid+length-1) = ...
cost_kid(label_kid:label_kid+length-1) + L_edges(label_edge);

% Then compare the pair of RR & RL, since they have the common
% right-edged projection which is to be added in the end.
label_parent = label_1(j-1,i_temp-1,i_temp-1); % Locate (2,-2,-2).
% Locate the edge from (i2,j) = (-2,2) which is the different part.
label_edge = label_3(i_temp-1,j-1); dir_temp = zeros(length,1);
dir_temp(length) = 4; % The last vertex (3,-1,3) always has dir=4.
% For RR, the parent vector (2,-2,-2)-(2,-2,0)-(2,-2,2) has the
% same length=3, while for RL, it is (2,-2,0)-(2,-2,2).
cost_temp = cost_parent(label_parent:label_parent+length-1) + ...
    R_edges(label_edge:label_edge+length-1); % RR
cost_temp1 = cost_parent(label_parent+1:label_parent+length-1)+ ...
    L_edges(label_edge+1:label_edge+length-1); % RL
for i = 1:length-1
    if cost_temp(i) <= cost_temp1(i);
        dir_temp(i) = 4;
    else
        cost_temp(i) = cost_temp1(i);
        dir_temp(i) = 3;
\end{verbatim}
% Needless to care about the last vertex (3,-1,3).
cost_temp = cost_temp + R_edges(label_edge);

% Lastly, cost_kid(label_kid:label_kid+length-1) & cost_temp:
for i = 1:length
    label_temp = label_kid+i-1;
    if cost_kid(label_temp) > cost_temp(i);
        cost_kid(label_temp) = cost_temp(i);
        dir(label_temp) = dir_temp(i);
    end
end

% The last column is just the NE corner vertex (3,3,3):
cost_kid(Vnum) = cost_parent(sum(1:j))+2*R_edges(label_3(j-1,j-1));
dir(Vnum) = 4;

cost_parent = cost_kid;
Vidx = fopen('pathIDX.txt', 'a+');fprintf(Vidx,'%d ',dir);fclose(Vidx);
end

%--------------------------------------------------------------------------
% Upper & Constrained
%--------------------------------------------------------------------------
for j = ss+1:K-1
    Vnum = (j+1)*(j+2)/2; cost_kid = zeros(Vnum,1); dir = ones(Vnum,1);
    % (j,-j,j)-...-(j,j-dist-2,j)-(j,j-dist,j)-...-(j,j,j)
% (j,-j,-j+dist+2) assume dist is an even number...
% |
% (j,-j,-j+dist) (j-1,1-j,-j+dist+1) from the former layer
% |
% ...
% |
% (j,-j,-j)
% Only need care about the smaller chopped triangular part:
% (j,-j,j)-...-(j,j-dist-1,j)
% |
% ...
% |
% (j,-j,-j+dist+1)
% Assign all the costs of the rest "inf" whose directions are needless.

% (4,-4, 4)---(4,-2, 4)---(4, 0, 4)---(4, 2, 4)---(4, 4, 4)
% \  \  \  \  \  /  /  /  /  /
% | (3,-3, 3) | (3,-1, 3) | (3, 1, 3) | (3, 3, 3)
% \  \  \  \  \  /  /  /  /  /
% (4,-4, 2)---(4,-2, 2)---(4, 0, 2)---(4, 2, 2)
% \  \  \  \  \  /  /  /  /  /
% | (3,-3, 1) | (3,-1, 1) | (3, 1, 1)
% \  \  \  \  \  /  /  /  /  say dist = 2
% (4,-4, 0)---(4,-2, 0)---(4, 0, 0)
% \  \  \  \  \  /  /  /  /  /
% | (3,-3,-1) | (3,-1,-1)
% The 1st column vector: (j,-j,-j+dist+1)-...-(j,-j,-j). Start with the
% (j-1,1-j,-j+dist) for dir = LR. For LL:
% (j,-j,j) (j-1,-j+1,j-1)->label_end
% | /
% (j,-j,j-2)
% |
% ... (j-1,-j+1,-j+dist+3)->label_start
% | /
% (j,-j,-j+dist+2)
% For LR:
% (j,-j,j)
% |
% ... (j-1,-j+1,j-1)->label_end
% |
% (j,-j,-j+dist+2)
% |
% (j-1,-j+1,-j+dist+1)->label_start-1

label_start = dist/2+2;
label_end = j;
range = label_end-label_start;
label_edge = label_3(-j+dist+3,j-1);
cost_kid(label_start:label_end) = cost_parent(label_start:label_end)...
+ L_edges(label_edge:label_edge+range);
cost_temp = cost_parent(label_start-1:label_end) + ...
R_edges(label_edge-1:label_edge+range);
for i = 0:range
  label_temp = label_start + i;
  if cost_kid(label_temp) > cost_temp(i+1);
    cost_kid(label_temp) = cost_temp(i+1);
    dir(label_temp) = 2;
  end
end
cost_kid(j+1) = cost_temp(range+2); dir(j+1) = 2;
cost_kid(label_start:label_end+1) = ...
cost_kid(label_start:label_end+1) + L_edges(label_3(-j+1,j-1));
for col = 2:j-dist/2-1
  % First, locate (4,-2,2):
  i1 = 2*(col-1)-j;
  i2 = i1+dist+2;
  label_kid = label_1(j,i1,i2);
  label_parent = label_1(j-1,i1+1,i2+1);
  label_edge = label_3(i2+1,j-1); % Find the edge from (3,-1,3).
  range = range-1;
  cost_kid(label_kid:label_kid+range) = ...
    cost_parent(label_parent:label_parent+range) + ...
    L_edges(label_edge:label_edge+range); % LL
  cost_temp = cost_parent(label_parent-1:label_parent+range) + ...
    R_edges(label_edge-1:label_edge+range); % LR
  for i = 0:range
    label_temp = label_kid + i;
  end
end
if cost_kid(label_temp) > cost_temp(i+1);
    cost_kid(label_temp) = cost_temp(i+1);
    dir(label_temp) = 2;
end
end
cost_kid(label_kid+range+1) = cost_temp(range+2);
dir(label_kid+range+1) = 2;
cost_kid(label_kid:label_kid+range+1) = ...
cost_kid(label_kid:label_kid+range+1) + ...
L_edges(label_3(i1+1,j-1));

label_parent = label_1(j-1,i1-1,i2-1);
label_edge = label_3(i2-1,j-1); % Find the edge from (3,-3,1).
dir_temp = zeros(range+2,1); dir_temp(range+2) = 4;
cost_temp = cost_parent(label_parent:label_parent+range+1) + ...
    R_edges(label_edge:label_edge+range+1); % RR
cost_temp1 = cost_parent(label_parent+1:label_parent+range+1) + ...
    L_edges(label_edge+1:label_edge+range+1); % RL
for i = 1:range+1
    if cost_temp(i) <= cost_temp1(i);
        dir_temp(i) = 4;
    else
        cost_temp(i) = cost_temp1(i);
        dir_temp(i) = 3;
    end
end
cost_temp = cost_temp + R_edges(label_3(i1-1,j-1));

for i = 1:range+2
label_temp = label_kid+i-1;
    if cost_kid(label_temp) > cost_temp(i);
        cost_kid(label_temp) = cost_temp(i);
        dir(label_temp) = dir_temp(i);
    end
end
end

label_NE_kid = label_1(j,j-dist-2,j);  % Locate (4,0,4).
label_NE_parent = label_1(j-1,j-dist-3,j-1);  % Locate (3,-1,3).
cost_kid(label_NE_kid) = cost_parent(label_NE_parent) + ... 
    R_edges(label_3(j-1,j-1)) + R_edges(label_3(j-dist-3,j-1));
dir(label_NE_kid) = 4;

    "inf" assignment, needless to care about the direction: 
i_stop = j+1-dist/2;
for col = 1:i_stop
    i_temp = j-2*(col-1);
    label_temp = label_1(j,-i_temp,-i_temp);
    cost_kid(label_temp:label_temp+dist/2) = inf*ones(dist/2+1,1);
end
% The "tail: part, a small triangle with consecutive labels:
label_tail = label_1(j,j-dist,j)+1;
cost_kid(label_tail:Vnum) = inf*ones(Vnum-label_tail+1,1);

cost_parent = cost_kid;
Vidx = fopen('pathIDX.txt', 'a+');fprintf(Vidx,'%d ',dir);fclose(Vidx);
end
for j = K:2*K-3-ss
    j_comp = 2*K-j-2; % j_complementary to locate "3" of (9,-3).
    Vnum = (j_comp+1)*(j_comp+2)/2;
    cost_kid = ones(Vnum,1)*inf; dir = ones(Vnum,1);
    % (8,-4, 4)---(8,-2, 4)---(8, 0, 4)---(8, 2, 4)---(8, 4, 4)
    % | \ /| \ /| \ /| \ / |
    % | (9,-3, 3) | (9,-1, 3) | (9, 1, 3) | (9, 3, 3)
    % | /\ /\ /\ /\ |
    % (8,-4, 2)---(8,-2, 2)---(8, 0, 2)---(8, 2, 2)
    % | \ /| \ /| \ /| \ / |
    % | (9,-3, 1) | (9,-1, 1) | (9, 1, 1)
    % | /\ /\ /\ /\ |
    % (8,-4, 0)---(8,-2, 0)---(8, 0, 0)
    % | \ /| \ /| \ / |
    % | (9,-3,-1) | (9,-1,-1)
    % | /\ /\ /\ /\ |
    % (8,-4,-2)---(8,-2,-2)
    % | \ /| \ /| \ / |
    % | (9,-3,-3)
    % | /|
    % (8,-4,-4)

    % Say dist = 2, then the triangle we are looking at is actually
    % (9,-3, 3)-(9,-1, 3)
    % | /|
    % (9,-3, 1)
\% (j,-j\textunderscore comp,j\textunderscore comp) \( (j-1,-j+1,j-1)\rightarrow\text{label\_end} \\
\% | \\
\% (j,-j\textunderscore comp,j\textunderscore comp-2) \\
\% | \\
\% ... \( (j-1,-j+1,-j+\text{dist}+3)\rightarrow\text{label\_start} \\
\% | \\
\% (j,-j\textunderscore comp,-j\textunderscore comp+\text{dist}+2) \\

label\_start = \text{dist}/2+2; \% =3 \\
label\_end = j\textunderscore comp+1; \% =4 \\
range = label\_end - label\_start; \% =1 \\
label\_parent\_LL = j\textunderscore comp+\text{dist}/2+4; \% Locate the label of \((8,-2,-2) = 8\) \\
label\_parent\_LR = label\_parent\_LL-1; \% Locate the label of \((8,-2, 0) = 7\) \\
label\_parent\_RL = \text{dist}/2+3; \% Locate the label of \((8,-4,-2) = 4\) \\
label\_parent\_RR = label\_parent\_RL-1; \% Locate the label of \((8,-4, 0) = 3\) \\

for col = -j\textunderscore comp:2:j\textunderscore comp-dist-2; \\
    i2 = col+\text{dist}+2; \% \((j,i1,i2) = (9,-3, 1)\) \\
    label\_edge = label\_4(i2,j,K); \% Caution: lower part is \(j\) not \(j\textunderscore 1\). \\

    cost\_kid(label\_start:label\_end) = ... \\
    cost\_parent(label\_parent\_LL:label\_parent\_LL+range) + ... \\
    L\text{\_edges}(label\_edge:label\_edge+range); \% LL \\
    cost\_temp = cost\_parent(label\_parent\_LR:label\_parent\_LR+range) + ... \\
    R\text{\_edges}(label\_edge:label\_edge+range); \% LR \\
for i = 0:range \\
    label\_temp = label\_start + i; \\
    if cost\_kid(label\_temp) > cost\_temp(i+1);
\begin{verbatim}
cost_kid(label_temp) = cost_temp(i+1);
dir(label_temp) = 2;
end

cost_kid(label_start:label_end) = ...
cost_kid(label_start:label_end) + ...
L_edges(label_4(col,j,K));

dir_temp = zeros(range+1,1);
cost_temp = cost_parent(label_parentRR:label_parentRR+range) + ...
    R_edges(label_edge:label_edge+range); \% RR
cost_temp1 = cost_parent(label_parentRL:label_parentRL+range) + ...
    L_edges(label_edge:label_edge+range); \% RL
for i = 1:range+1
    if cost_temp(i) <= cost_temp1(i);
        dir_temp(i) = 4;
    else
        cost_temp(i) = cost_temp1(i);
        dir_temp(i) = 3;
    end
end
cost_temp = cost_temp + R_edges(label_4(col,j,K));

for i = 1:range+1
    label_temp = label_start+i-1;
    if cost_kid(label_temp) > cost_temp(i);
        cost_kid(label_temp) = cost_temp(i);
        dir(label_temp) = dir_temp(i);
    end
\end{verbatim}
end

label_start = label_start+j_comp+1-(col+j_comp)/2;  \% =7
range = range-1;  \% =0
label_end = label_start+range;  \% =7
label_parentLL = label_parentLL+j_comp+1-(col+j_comp)/2;  \% =12
label_parentLR = label_parentLL-1;  \% =11
label_parentRL = label_parentRL+j_comp+2-(col+j_comp)/2;  \% =9
label_parentRR = label_parentRL-1;  \% =8
end

cost_parent = cost_kid;
Vidx = fopen('pathIDX.txt', 'a+');fprintf(Vidx,'%d ',dir);fclose(Vidx);
end

\%-------------------------------------------------------------------------------------
\% Lower & Unconstrained
\%-------------------------------------------------------------------------------------
    j_comp = 2*K-j-2;  \% j_complementary to locate "2" of (10,-2).
    Vnum = (j_comp+1)*(j_comp+2)/2;
    cost_kid = zeros(Vnum,1); dir = ones(Vnum,1);
    \% (9,-3, 3)---(9,-1, 3)---(9, 1, 3)---(9, 3, 3)
    \%  \|  \|  \|  \|   /
    \%  |(10,-2, 2) |(10, 0, 2) |(10, 0, 2)
    \%  \|  \|  \|  \|
    \% (9,-3, 1)---(9,-1, 1)---(9, 1, 1)
    \%  \|  \|  
71
\% label\_start = 1;
label\_end = j\_comp+1; \% =3
range = label\_end - label\_start; \% =2
label\_parentLL = j\_comp+3; \% Locate the label of \((9,-1,-1) = 5\)
label\_parentRL = 2; \% Locate the label of \((9,-3,-1)\)
label\_parentRR = 1; \% Locate the label of \((9,-3,-3)\)

for col = -j\_comp:2:j\_comp;
    label\_edge = label\_4(col,j,K); \% \((j,i1,i2) = (j,col,col) = (10,-2,-2)\)

    cost\_kid(label\_start:label\_end) = ...
    cost\_parent(label\_parentLL:label\_parentLL+range) + ...
    L\_edges(label\_edge:label\_edge+range); \% LL
    cost\_temp = cost\_parent(label\_parentLL:label\_parentLL+range-1) + ...
    R\_edges(label\_edge+1:label\_edge+range); \% LR - Caution!

    for i = 1:range \% From \((10,-2,0)\) to \((10,-2,2)\)
        label\_temp = label\_start + i;
        if cost\_kid(label\_temp) > cost\_temp(i);
            cost\_kid(label\_temp) = cost\_temp(i);
            dir(label\_temp) = 2;
        end
    end
end
cost_kid(label_start:label_end) = ...
    cost_kid(label_start:label_end) + ...
    L_edges(label_4(col,j,K));

dir_temp = zeros(range+1,1);

cost_temp = cost_parent(label_parentRR:label_parentRR+range) + ...
    R_edges(label_edge:label_edge+range); % RR

cost_temp1 = cost_parent(label_parentRL:label_parentRL+range) + ...
    L_edges(label_edge:label_edge+range); % RL

for i = 1:range+1
    if cost_temp(i) <= cost_temp1(i);
        dir_temp(i) = 4;
    else
        cost_temp(i) = cost_temp1(i);
        dir_temp(i) = 3;
    end
end

cost_temp = cost_temp + R_edges(label_4(col,j,K));

for i = 1:range+1
    label_temp = label_start+i-1;
    if cost_kid(label_temp) > cost_temp(i);
        cost_kid(label_temp) = cost_temp(i);
        dir(label_temp) = dir_temp(i);
    end
end

label_start = label_start+j_comp+1-(col+j_comp)/2; % =4
range = range-1; \% =1
label_end = label_start+range; \% =5
label_parentLL = label_parentLL+j_comp+1-(col+j_comp)/2; \% =8
label_parentRL = label_parentRL+j_comp+2-(col+j_comp)/2; \% =6
label_parentRR = label_parentRL-1; \% =5
end

cost_parent = cost_kid;
Vidx = fopen('pathIDX.txt', 'a+');fprintf(Vidx,'%d ',dir);fclose(Vidx);
end

% %--------------------------------------------------------------------------
% % Path Recovery
% %--------------------------------------------------------------------------
% % First of all, the 3D path is a (2*K-1) by 3 matrix.
% Vidx = fopen('pathIDX.txt', 'r');
% [dir,V_num_total] = fscanf(Vidx,'%d ');
% fclose(Vidx);

% pathV = zeros(destlevelup+1,3); \% From 0th layer to destlevelup-th.
% pathV(destlevelup+1,:) = [destlevelup,desthorizon,desthorizon];
% % First, we allocate the destination’s label in dir, i.e. cumulative label:
% V_num_cml_temp = V_num_total - sum(1:2*K-1-destlevelup);
% label_dest_cml = V_num_cml_temp + label_dest;
% dir_temp = dir(label_dest_cml);

% for j_cpl = K:destlevelup \% j_complementary
% \% j = K + destlevelup - j_cpl;
% switch dir_temp

74
case 1 % LL
   pathV(j,:) = pathV(j+1,:)+[-1, 1, 1];
   case 2 % LR
   pathV(j,:) = pathV(j+1,:)+[-1, 1,-1];
   case 3 % RL
   pathV(j,:) = pathV(j+1,:)+[-1,-1, 1];
   case 4 % RR
   pathV(j,:) = pathV(j+1,:)+[-1,-1,-1];
end
V_num_cml_temp = V_num_cml_temp - sum(1:2*K-j);
label_cml_temp = V_num_cml_temp + ...
    label_2(pathV(j,1),pathV(j,2),pathV(j,3),K);
dir_temp = dir(label_cml_temp);
end

def case 1 % LL
   pathV(j,:) = pathV(j+1,:)+[-1, 1, 1];
   case 2 % LR
   pathV(j,:) = pathV(j+1,:)+[-1, 1,-1];
   case 3 % RL
   pathV(j,:) = pathV(j+1,:)+[-1,-1, 1];
   case 4 % RR
   pathV(j,:) = pathV(j+1,:)+[-1,-1,-1];
end
V_num_cml_temp = V_num_cml_temp - sum(1:j);
label_cml_temp = V_num_cml_temp + ...
function num_1 = label_1(j,i1,i2)
% Coordinate -> num in each layer of the 3D graph,
% i.e. the label in cost_parent, e.g.
% v4-v7-v9-v10  (3,-3, 3)-(3,-1, 3)-(3, 1, 3)-(3, 3, 3)
% | | | /     | | | |
% v3-v6-v8    (3,-3, 1)-(3,-1, 1)-(3, 1, 1)
% | | / <- | | / 
% v2-v5       (3,-3,-1)-(3,-1,-1)
% | /     |   / 
% v1        (3,-3,-3)
num_1 = (i1+j)/2*(j+1)+(i2+j)/2+1-sum(1:(i1+j)/2);
end

function num_2 = label_2(j,i1,i2,K)
% The lower-triangular version of function label_1:
j_temp = 2*K-2-j;
num_2 = (i1+j_temp)/2*(j_temp+1)+(i2+j_temp)/2+1-sum(1:(i1+j_temp)/2);
end

function num_3 = label_3(i,j)
% Given a coordinate, how to find the 2 edges strating from this vertex,
% but this only works for the upper triangle:
% v1=(0,0)
function num_4 = label_4(i,j,K)
% Lower triangular version of function label_3: j > K
% num_temp = 2*K-2-j;
% num_4 = K*K-(num_temp-(i+num_temp)/2)-sum(1:num_temp);
num_4 = K-K*K+(i+j)/2+(4*K-3-j)*j/2;
% The difference between label_3 & label_4 is: label_3 uses the former
% layer (j-1)-th to label while label_4 uses the current layer j-th to
% label. e.g. when you are computing the 3rd layer v7-...-v10, you are
% using cost_parent(2nd layer)+label_3(2nd layer), but when you are
% computing the 4th layer v11-...-v13, you are using
% cost_parent(3rd layer)+label_4(4th layer)
end

% function num_2 = label_2(j,i1,i2)
% % Since L_edges & R_edges are 2 arrays, we need the 'accumulated version'
% % of 'label function' to locate the coordinate.
% num_2 = (i1+j)/2*(j+1)+(i2+j)/2+1-sum(1:(i1+j)/2)+...
% (j*(j+1)*(2*j+1)/6+(j+1)*j/2)/2;
% end

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>0/</td>
<td>\0</td>
</tr>
<tr>
<td>(1,-1)</td>
<td>(1,1)</td>
<td>0/</td>
</tr>
<tr>
<td>(2,-2)</td>
<td>(2,0)</td>
<td>(2,2)</td>
</tr>
<tr>
<td>(3,-3)</td>
<td>(3,-1)</td>
<td>(3,1)</td>
</tr>
<tr>
<td>(4,-4)</td>
<td>(4,-2)</td>
<td>(4,0)</td>
</tr>
<tr>
<td>(5,-5)</td>
<td>(5,-3)</td>
<td>(5,-1)</td>
</tr>
<tr>
<td>(6,-6)</td>
<td>(6,-4)</td>
<td>(6,-2)</td>
</tr>
<tr>
<td>(7,-5)</td>
<td>(7,-3)</td>
<td>(7,-1)</td>
</tr>
</tbody>
</table>
A.3 compare3D.m

function [t1,t2,t3] = compare3D(K)

    num_edge = K*(K-1); cost_L = rand(1,num_edge); cost_R = rand(1,num_edge);
    tic;
    cvx_begin
    variable x(num_edge)
    variable y(num_edge)
    minimize( cost_L * x + cost_R * y )
    subject to
    % Integrality
    for i = 1:num_edge
        0 <= x(i) <= 1;
        0 <= y(i) <= 1;
    end
    % Corner point
    x(1) + y(1) == 2; % Source
    x(num_edge) + y(num_edge) == 2; % Sink
    cvx_end
x(K-1) == y(num_edge-K+2);
y(K-1) == x(num_edge-K+2);

% Edge point
for i=1:K-2
    x(i) == x(i+1) + y((K-1)*i+1);
y(i) == y(i+1) + x((K-1)*i+1);
x(num_edge-i+1) == x(num_edge-i) + y(num_edge-(K-1)*i);
y(num_edge-i+1) == y(num_edge-i) + x(num_edge-(K-1)*i);
end

% Interior point
for i1 = 1:K-2
    for i2 = 1:K-2
        num_temp_x = (K-1)*i1 + i2;
        num_temp_y = (K-1)*i2 + i1;
        x(num_temp_x) + y(num_temp_y) == ...  
              x(num_temp_x+1) + y(num_temp_y+1);
    end
end

% dist = 2
for i1 = 1:K-2
    for i2 = 1:K-2
        num_temp_x = (K-1)*i1 + i2;
        num_temp_y = (K-1)*i2 + i1;
        x(num_temp_x) + y(num_temp_y) <= 1;
        x(num_temp_x) + y(num_temp_y+1) <= 1;
        x(num_temp_x+1) + y(num_temp_y) <= 1;
        x(num_temp_x+1) + y(num_temp_y+1) <= 1;
    end
end
cvx_end

t1 = toc;

Upper = K*(K+2)*(K+1)/6; Total = 2*Upper - sum(1:K);

tic; [Hash,CE,M] = DiamondAdj(K,cost_L,cost_R,4,2); t2 = toc;
tic; [C,P,pred] = graphshortestpath(M,1,Total); t3 = toc;

% tic; [cost_parent] = path3D(K,cost_L',cost_R',2,4,Total); t3 = toc;
% t3 = 2*t3;

% Plot
N = length(P); P1 = zeros(N,2); P2 = P1;
for i = 1:N
    P1(i,:) = [Hash(P(i),1),Hash(P(i),2)];
    P2(i,:) = [Hash(P(i),1),Hash(P(i),3)];
end
for i = 1:N-1
    plot([P1(i,1),P1(i+1,1)], [P1(i,2),P1(i+1,2)], 'g'); hold on;
    plot([P2(i,1),P2(i+1,1)], [P2(i,2),P2(i+1,2)], 'b'); hold on;
end

end