Optimization of Cable Cycles: A Trade-off between Reliability and Cost

Master Thesis in Micro-data Analysis

Author:
Barsam Payvar

Supervisors:
Hasan Fleyeh
Pascal Rebreyend

March 2014

Business Intelligence Program
School of Technology and Business Studies
Dalarna University
Abstract

In this thesis, a problem in routing a cable cycle for linking a set of devices is addressed. A hybrid method for an efficient search for a reasonable solution is proposed as well. To assess candidate cable cycles, risk and length factors are defined based on existing requirements. Due to a trade-off which often exists between the risk and length while deciding a solution, a criterion for ranking candidates is defined. To have the search efficient, a set of techniques is proposed to reduce the search-space. For that, heuristics and meta-heuristics are employed to avoid high cost or lengthy-risky solutions. Moreover, the graph which is formed from a raw data is labelled with conditional direction signs in order to have the search navigated toward feasible and worthy solutions. The result shows the method provides solutions which are both technically and financially reasonable. Further, it is proven the techniques are very effective in reducing the computational time of the search to a reasonable amount.

**Keywords:** Combinatorial Optimization, Cable Routing Problem, Self-Healing Ring Network, Risk of Failure, Heuristic Estimation
Acknowledgments

My special thanks go to my parents who have patiently supported and encouraged me during not only this work but whole my life. I wish to express my gratitude to my supervisors Dr. Hasan Fleyeh and Dr. Pascal Rebreyend for all their helpful remarks, suggestions and constructive criticisms. I must gratefully acknowledge the help of Mats Karlsson at AB Tändkulan by providing data and dedicating time as well as giving useful feedbacks. I also would like to appreciate Dr. Siril Yella and other staff members of the School of Technology and Business Studies who helped me during my studies at Dalarna University. After all, I offer my regards to friends and all who directly or indirectly gave me an assist.
## Contents

List of Figures ......................................................................................................................... VI  
List of Tables .............................................................................................................................. VIII  
1 Introduction .......................................................................................................................... 1  
  1.1 Literature Review ......................................................................................................... 3  
  1.1.1 Literatures Considering Ring Network Design Problems ........................................... 3  
  1.1.2 A Literature Considering the Selective Travelling Salesman Problem ....................... 5  
  1.1.3 Literatures Considering the Steiner Travelling Salesman Problem ............................... 5  
  1.2 Aim and Scope of the Work ............................................................................................ 6  
2 Technical Details and Modelization ...................................................................................... 1  
  2.1 Reliability of a Cycle ...................................................................................................... 1  
  2.1.1 Cases in Which It Is Impossible to Link All Objects with a Perfect Cycle ................. 2  
  2.1.2 Cases in Which a Perfect Cycle Linking All Objects Is Not Preferred ....................... 3  
  2.2 Differences between Shortest, Most Reliable, Ideal and Most Reasonable Cycles ....... 3  
  2.3 Modelization ................................................................................................................ 4  
3 Data Description .................................................................................................................. 5  
  3.1 Set of Routes ................................................................................................................ 5  
  3.2 Set of Objects ................................................................................................................. 6  
  3.3 Set of Levels .................................................................................................................. 7  
4 Problem Formulation ........................................................................................................ 9  
5 Data Transformation .......................................................................................................... 13  
  5.1 Route Orientation ....................................................................................................... 13  
  5.2 Object Access Point ..................................................................................................... 14  
  5.3 Graph Construction ..................................................................................................... 15  
6 Methods ............................................................................................................................... 18  
  6.1 Issues for Developing Approximate Algorithms ............................................................ 18  
  6.2 A Method Based on Finding Perfect Cycles .................................................................... 21  
  6.3 Proposed Method ......................................................................................................... 22  
    6.3.1 Techniques for Making a Brute-Force-Search-Space Smaller .................................. 22  
    6.3.2 Overview of the Method ......................................................................................... 24  
    6.3.3 Adding Signs to the Graph .................................................................................... 25  
    6.3.4 Finding Near the Shortest Cycle ............................................................................ 33
List of Figures

Figure 1.1 A photo taken from: (a) a cabinet or primary object; (b) an ordinary object; (c) a ladder or route ................................................................. 2
Figure 2.1 Illustration of a perfect cycle (a) in a normal condition and (b) after a cut .......... 1
Figure 2.2 Illustration of a non-perfect cycle (a) in a normal condition and (b) after a cut .... 2
Figure 2.3 Example of two cases in which (a) an object located on a dead-end and (b) two parallel objects make it impossible to have a perfect cycle solution ........................................ 2
Figure 2.4 A case when (a) a non-perfect cycle solution could be shorter than (b) a perfect cycle solution .................................................................................. 3
Figure 2.5 A length-risk plot for showing the area which is searched for a solution .......... 4
Figure 3.1 Visualized example of a three-route structure (visualization of Table 3.1) .......... 6
Figure 3.2 Visualization of a 78-route structure of a real project (a power plant) .......... 6
Figure 3.3 Partitioning a non-rectangular level into three rectangular sublevels ................. 7
Figure 3.4 Indicating a rectangular sublevel by coordinates of two opposite corners .......... 8
Figure 4.1 Illustration of (a) routes and object as well as (b) the corresponding graph ........ 9
Figure 4.2 Example of a valid cycle with three risky edges .............................................. 11
Figure 5.1 Structure of the required data and information ................................................. 13
Figure 5.2 Procedure of building a graph ........................................................................ 17
Figure 6.1 Transformation of (a) an original graph of the problem to (b) a Hamiltonian graph based on the shortest path between the required nodes ........................................ 19
Figure 6.2 Example of the method of finding and merging perfect cycles ......................... 22
Figure 6.3 Example of a depth-first search in a tree .......................................................... 23
Figure 6.4 Example of type 1 signs .................................................................................. 26
Figure 6.5 Procedure of signing an undirected graph with type 1 signs ............................ 26
Figure 6.6 Example of (a) a valid cycle with a return at an optional node and (b) a resulted valid cycle by removing the return ......................................................... 28
Figure 6.7 Two different valid cycles with the same common paths on a branch .............. 29
Figure 6.8 Illustration of different types of the nodes on a branch: (a) left to right movement; (b) right to left movement .............................................................................. 30
Figure 6.9 Example of a branch with three required nodes: (a) conditional directions; (b) partial valid cycle excluding $e_{bg}$; (c) partial valid cycle excluding $e_{mbg}$ .................................................. 31
Figure 6.10 Example of an invalid cycle resulted by an early return in a dead-end path ...... 32
Figure 6.11 Example of a weighted graph with one primary, two required and four optional nodes ......................................................................................... 34
Figure 6.12 Examples of invalid cycles with (a) characteristic 1, (b) characteristic 2 and (c) characteristic 3 .............................................................................. 36
Figure 6.13 Measuring Manhattan distances for heuristic estimation of the final length of a partial solution .............................................................................. 38
Figure 7.1 Illustration of changes in length and risk with most reasonable cycles in comparison with corresponding near the shortest cycles (illustration of column 2-3 of Table 7.2) .................................................................41
Figure 7.2 Illustration of changes in length and risk with most reliable cycles in comparison with corresponding near the shortest cycles (illustration of column 4-5 of Table 7.2) ............42
Figure 7.3 Illustration of time taken for obtaining solutions (most reasonable and reliable) by exploiting all and not exploiting each of the proposed techniques .........................................................44
List of Tables

Table 3.1 Example of a three-route structure dataset .................................................................5
Table 6.1 Example of a shortest distance matrix ........................................................................34
Table 6.2 Example of a shortest path array ..............................................................................35
Table 7.1 Eleven problem instances and properties of three different types of solution to them
..................................................................................................................................................40
Table 7.2 Changes in length and risk with most reasonable and most reliable cycles in
comparison with corresponding near the shortest cycles ..........................................................41
Table 7.3 Run-times of obtaining three different types of solutions to 11 problem instances by
using all proposed techniques .......................................................................................................43
1 Introduction

The problem this thesis addresses could be categorized as a network design problem. Particularly, it is a cable routing problem with specific properties, but in general, it is a routing problem which has some common properties with routing problems in transportation, logistics, warehousing or other fields. Therefore, the techniques, which are introduced in this thesis, could be useful in solving other routing or network design problems.

Regardless of similarities or dissimilarities between routing problems, they are often in the class of intractable combinatorial problems. By intractable it is referred to those problems which are so hard to be possibly solved by a polynomial time algorithm (according to Garey et al. [1]). Some of those problems has been proven to be NP-hard and are reducible to the current problem (e.g., the problem of order-batching in a warehouse [2]). Accordingly, one can conclude the current problem is NP-hard and solving its large instances is impractical by a naive brute-force search. To be familiar with the problem and its hardness, first, the origin of the problem is stated as follows.

In fact, the problem was arisen during projects undertaken by AB Tändkulan1, a company working in industrial process automation consultancy in Sweden. The company has customers from various industries such as energy & environment, food, iron & steel, paper, mining and timber; and there is variety of activities in their projects.

In many of the projects, some devices (called objects) are installed for control or management purposes at the site of a customer. The objects are fixed in certain places of a building and planned to be connected together by cables. There are also ladders (see, Figure 1.1c) which are fixed in many places of the building, near the ceilings or the walls (horizontally or vertically), to support the cables. The cables are only routed through the ladders and tied on them (so the ladders are called routes in this work).

There might be many possibilities to route a cable through the routes for connecting a set of objects. Among all the possible paths, the company chooses one based on two concerns. One concern is financial and related to the length of the cable. The other is technical and related to the reliability of the path in case of a cut. In many cases, taking a shorter path could be achieved in return for a lower reliability and vice versa. This means, in those cases, the company has to compromise on the path.

Also, considering all the available paths manually is often difficult or impossible. That is the point the company aimed to automate the job by means of computer. Therefore, they collaborated with Farooq [3] to provide a tool for that purpose. Although the result could

---

1 http://www.abtk.se/
satisfy the company from the financial point of view, there were some technical issues with the solutions provided by the proposed method. Hence, the current work started for solving the addressed issues and providing a tool for meeting the company’s requirements.

To this end, some new methods were developed for routing the cable. Those methods will be discussed in this work as well as the results. In the following, an overview of the problem which is needed for further discussion will be given.

First, let us start by more details about the objects and the way they are connected. In a set of objects which are linked by a same cable together, one of the objects is a cabinet like the one in Figure 1.1a and called primary object. A primary object plays a vital role in the operation of the rest of the objects which are called ordinary objects (see, Figure 1.1b). A connecting cable forms a cycle through which every object is linked with exactly two other objects. The aim of such connectivity is to avoid losing an object in case of a cut in any part of the cable path. That is, if a link between two objects is disconnected for any reason, all the ordinary objects would still accessible from the primary object. Therefore, there would not be a serious problem in the system until the cable is repaired. However, the problem would be serious when multiple links are disconnected at the same time. In that case, the whole cable would be divided into two separate parts. One part would be linking the primary object and possibly some of the ordinary objects, while the second part would be linking the rest of the ordinary objects. Since there would be no access from the primary object to the objects linked by the second part, those objects would be lost and that would be troublesome. The more the number of objects connected by the second part, the more trouble one faces in that situation.

Now, one might ask why two or more links might be disconnected at the same time, or how much it is probable to have such a problem. First remember, a cable has to be routed through the fixed routes. Hence, the access to an object and routing a cable is restricted. That might

Figure 1.1 A photo taken from: (a) a cabinet or primary object; (b) an ordinary object; (c) a ladder or route
cause a connecting cable to be routed through certain routes (or part of them) twice, in some cases. This means, some parts of the cable are placed next to each other. Since a cut is often due to external factors such as a sharp object or fire, two parts of the cable with the same path are very probable to be cut at the same time. Thus, a path shared between two parts of a cable is risky, and the longer the length of the path, the higher the risk of facing a trouble.

A risky path for a cable might be taken either inevitably or intentionally for the sake of a path shortening. Since risky paths reduce the reliability of the system, it should be decided whether they are worth to take in return for a shorter path. To make such a decision, a path should be evaluated by measuring its length and the risk of taking the path. However, the lack of such a measure for the risk (not defined by the company), is addressed as a problem in evaluating a path. As soon as the measure is defined for the risk, a next issue would be the identification of all the feasible paths for taking one as a solution. That job is often complex and very time-consuming even by means of computer and that is what makes the problem intractable (i.e. NP-hard).

The mentioned issues are the main ones in solving the problem and will be discussed more in the work gradually. There are also other issues which will be addressed in their right place in the coming chapters.

1.1 Literature Review

Since, according to the author’s study, the current problem has not previously been addressed by others, here, some of the works which are partly related are mentioned. The order of review is as follows. First, the thesis by Farooq [3] which is a relaxed version of the current problem will be reviewed. Then, some of the other works addressing ring network design problems with different configurations and constraints will be inspected. In addition, since the current problem has similarities with the Selective Travelling Salesman Problem and the Steiner Travelling Salesman Problem, some literatures relevant to those problems will be reviewed as well.

1.1.1 Literatures Considering Ring Network Design Problems

As mentioned earlier, Farooq [3] tackled a similar cable routing problem of the Tändkulan Company as a thesis. Since the only factor in his work was the length of the cable, he transformed the problem into the Travelling Salesman Problem (TSP) (see, Appendix A). He tested three different algorithms in finding a shortest/near shortest cable cycle for linking a given set of objects through available routes. He employed greedy, simulated annealing (see, Appendix B) and exhaustive search algorithms as well as comparing the results together. He came to the decision that the simulated annealing algorithm finds a good solution (i.e., near the shortest solution) better and faster than the other two.
The work by Luss et al. [4], besides its main objective which is solving a particular network design problem, is helpful in understanding the way self-healing ring networks work. In that work, the authors explained how rings with specific components provide restoration of service within a short time in case of a single node or link failure (similar to what explained earlier). Also, they presented a heuristic approach for designing networks comprised of interconnected rings. The design is based on existing fiber links and demands between nodes which are going to form the network. The problem is to find an optimal subset of rings such that:

- every ring sharing at least two nodes with another ring;
- nodes in a ring being distinct;
- all the nodes being covered;
- no new fiber link being added;
- the total cost being minimized.

There are also some restrictions on the number of nodes in each ring and the lengths of links and rings. For solving the problem, the authors assumed that most telecommunications networks are relatively sparse (i.e. the nodes are not fully pairwise connected) which is also true for the current problem. Therefore they suggested, first, to generate a complete list of candidate rings by a procedure (not clear in their work) that finds cycles in a graph. Then, a set covering algorithm is used to select a subset of rings with a minimum cost such that every node is covered at least by one ring. Finally, some additional rings are selected as bridges, in case there are rings not dual interconnected. The result shows, their solution approach is applicable for problems with the same configuration and constraints where the network is quite sparse.

Wasem [5] proposed two algorithms for solving another ring network design problem. The problem is to find a shortest cycle for connecting a required subset of offices (other offices are optional to be included) using available links. The constraints in solving the problem are as below:

- the length of paths should not exceed a preset threshold;
- the number of optional offices on the path between two required offices should not exceed certain value;
- the offices included in a solution should be distinct.

Due to the constraints, the algorithms may fail in finding a solution even if a cycle which links all of the required offices exists. The two proposed algorithms are complementary of each other; if the simple one fails, the other one is employed. The first algorithm tries to find two link-disjoint shortest paths between a main required office and another required office furthest from it by using the Dijkstra’s shortest path algorithm (see, Appendix C). If the cycle formed by the two paths contains all of the required offices, it is accepted as a solution;
otherwise the second algorithm begins as follows. First, all of the paths between all pairs of required offices (containing only two ring offices and considering the mentioned constraints) are identified by a depth-first search (for DFS see Subsection 6.3.1). Then, combinations of node disjoint paths are tried to build up a ring. The algorithm starts at a required office and picks the shortest path which goes to another required office. Next, it picks the shortest node disjoint path out of the latest office. It is continued in this way until either a ring is formed or no more paths from the last node could be picked. In the later case the algorithm backs up and tries another combination of paths. In case of not finding any ring, after trying all the combination of paths, the mentioned threshold is increased. However, due to technical restrictions, rings more than certain length are not acceptable and the algorithm may fail.

Wasem [5] also discussed dissimilarities of the problem with the TSP. Since the addressed dissimilarities are some of those between the current problem and the TSP, they are mentioned briefly as follows:

- The classic TSP deals with a complete graph, while in most variation of ring routing problem the graph is sparse.
- A cycle or tour in the TSP includes all the nodes of graph, whereas in most ring routing problems, some of the nodes are only required (the rest optional).
- In the TSP, the objective is to find a tour with a minimum cost (e.g., length), but that type of solution may not be technically appealing in a ring routing problem.

Fink et al. [6] studied the similarities in different ring network design problems and produced a general problem formulation. They also presented the application of meta-heuristics to some ring network design problems. They claimed that the General Ring Network Design Problem (GRNDP) covers a great variety of combinatorial optimization problems with a ring-like structure. However, there are many other ring network design problems that were not considered in their work.

1.1.2 A Literature Considering the Selective Travelling Salesman Problem

Laporte et al. [7] proposed a method for solving the Selective Travelling Salesman Problem. The problem is maximization of total profit with cost (i.e., length) lower than a preset value. Every inclusion of the nodes in the cycle solution increases the total profit, but it can also increase the cost. The approach for solving the problem consists of finding lower and upper bounds by approximate algorithms and exploiting them in an exact algorithm using branch-and-bound (BB – see, Subsection 6.3.1).

1.1.3 Literatures Considering the Steiner Travelling Salesman Problem

Cornuéjols et al. [8] consider a variant of the classical TSP calling “the Steiner Travelling Salesman Problem”. The goal in Steiner TSP, same as the classical TSP, is minimization of
the tour length, but visiting some of the nodes is not mandatory. Moreover, unlike the classical TSP, nodes and links could be included in a tour more than once. There are also problems which are classified as a Steiner TSP, such as the problem of order-picking in a warehouse [9][10][11] and the Steiner Ring Network Design Problem [12].

As stated in [8][13][14], an instance of the Steiner TSP could be transformed into an instance of the standard TSP by calculating the shortest path between every pair of required nodes. Other than the disadvantages of the transformation (see, [8]), it is not applicable for some variants of the Steiner TSP (e.g., the current problem – See, Section 6.1).

1.2 Aim and Scope of the Work

According to the need of the Tändkulan Company for automation of the whole process of the cable routing design by a computer application, the work was divided into two main parts:

- Research and development of proper methods and algorithms for cable routing
- Design and development of an interactive and user-friendly GUI (Graphical User Interface) equipped with tools for data visualization, adjustment of the user need and feeding/exporting data in a convenient way

Due to the different contexts of the two parts and the importance of the first part in combinatorial optimization, this thesis only considers the algorithms and techniques developed for the cable routing. To this end, the scope is limited to:

- studying whether or not an approximate algorithm could be employed or developed in order to find a near optimal solution;
- making attempts on exact algorithms with and without using improvement techniques (existing or new) which makes the search-space smaller, and then comparing the results to see the differences;

The remaining chapters are organized as follows. Chapter 2 discusses the technical issues and essential concepts about the ring architecture in more details as well a model for solving the problem. Chapter 3 describes the data. Chapter 4 provides formulation for the problem. Chapter 5 describes the methods for extracting desired information from the raw data and forming a graph for the problem. Chapter 6, first, investigates the issues in developing an approximate algorithm for solving the problem. The rest of Chapter 6 explains other methods which were tried (but failed) as well as the final method. Chapter 7 presents and discusses the result. Finally, Chapter 8 gives a conclusion.
2 Technical Details and Modelization

In the previous chapter, it was shortly described how ring networks (a cyclic connectivity) reduce troubles caused by a link or node failure. Also, possible problems in forming a perfect cycle (see the following section) were briefly pointed out. In this chapter, first, the mentioned subjects will be elaborated. Then, different types of cycles which are considered in this work will be defined. At the end of this chapter, a model for the problem will be presented for the sake of clarity.

2.1 Reliability of a Cycle

As stated before, one reason in forming a cycle for connecting a set of objects is to provide system restorability for increasing the reliability of a network. However, the amount of restorability of a system (in case of a failure) could differ depending on the path of cycle. To expand this further, first remind that a cycle (in this work) is a closed path through available routes which connects a set or subset of objects. Links between the objects could be joined in some parts of the path, but they are not preferred. Based on that, cycles are divided into two general types by means of graph definitions (see, Chapter 4), as follows:

- A perfect cycle: a cycle including distinct edges
- A non-perfect cycle: a cycle which is not perfect (i.e., includes indistinct edges)

For a given problem, a perfect cycle connecting all of the objects (if any) is an ideal solution technically, because it is completely reliable. By such a solution, a physical damage in any part of the cable path will result in a single link failure. In that case, the system will restore itself, since no object will be disconnected from the primary object (see, Figure 2.1).

![Figure 2.1 Illustration of a perfect cycle (a) in a normal condition and (b) after a cut](image)

On the contrary, with a non-perfect cycle as a solution, there would be the risk of losing two links at the same time. In that case, some of the objects will be lost, because there will be no

links between them and the primary object (see, Figure 2.2). Therefore, a non-perfect cycle as solution is less reliable and not very desirable technically.

![Diagram of non-perfect cycle solution](image)

**Figure 2.2 Illustration of a non-perfect cycle (a) in a normal condition and (b) after a cut**

Now, an important question arises here that: why should a non-perfect cycle be considered as a candidate in spite of higher risk of failure?

The direct answer to the question is that, in some graphs, forming a perfect cycle which connects all of the objects might be either (1) impossible or (2) very costly (i.e., very long). To clarify this, the two cases will be elaborated in the following subsections.

### 2.1.1 Cases in Which It Is Impossible to Link All Objects with a Perfect Cycle

As mentioned above, in some graphs there is no way to connect all of the objects with a perfect cycle. This issue arises in each of the following cases:

1. Where there is at least one object with one-way accessibility (i.e., an object located on a dead-end; see, Figure 2.3a)
2. Where two or more objects are parallel with the primary object (see, Figure 2.3b)

![Diagram of two cases](image)

**Figure 2.3 Example of two cases in which (a) an object located on a dead-end and (b) two parallel objects make it impossible to have a perfect cycle solution**
2.1.2 Cases in Which a Perfect Cycle Linking All Objects Is Not Preferred

In many cases linking all of the objects by a non-perfect cycle could be done in a shorter way than by a perfect cycle (like the ones in Figure 2.4). Due to financial reasons, the company sets a length limit to accept solutions which are more reliable than the shortest one. Therefore, a perfect cycle solution longer than the length limit is rejected.

![Diagram showing a perfect cycle linking all objects](image)

Figure 2.4 A case when (a) a non-perfect cycle solution could be shorter than (b) a perfect cycle solution

2.2 Differences between Shortest, Most Reliable, Ideal and Most Reasonable Cycles

There are also other types of cycle which are referred to in this work based on their properties (i.e., length and risk) and are defined as follows:

- **Shortest cycle**: As its name suggests, it is a cycle which connects all of the objects in the shortest possible way with any risk. A shortest cycle is optimal from an economical point of view.

- **Most reliable cycle**: It is a cycle which connects all of the objects with the minimum risk of failure and as short as possible. Note that the risk of a most reliable cycle is not necessarily zero (i.e., 100% reliable), but sometimes it is. A most reliable cycle is optimal from a technical point of view.

- **Ideal cycle**: It is a desired case when the shortest and most reliable cycles are identical. However, in many cases it may not exist.

- **Most reasonable cycle**: It is a cycle which connects all of the objects with properties as close as possible to an ideal cycle. There is also a parameter, called *acceptable extra length* (determined by the company), which does not allow that the length of a most reasonable cycle to be more than some amount far from the shortest cycle’s length. Also, obviously, the risk of a most reasonable cycle cannot be more than the shortest cycle’s risk.

The model in the next section makes the above definitions more clear.
2.3 Modelization

Although the problem formulation in Chapter 4 accurately defines the properties of a most reasonable solution, for now, a model based on the preceding definitions might be helpful. The plot in Figure 2.5 depicts the model by which the area of the search for a most reasonable solution is defined.

Assuming a search for a solution, the blue dashed line in the figure would be an indicator for the properties (i.e., length and risk) of current best found solutions. Slope of the blue line which is “-1” means all candidates being indicated by the line are at the same level of importance as a current solution. At the beginning of the search, the blue line would lie upon the black dashed line which is crossing the red dot (i.e., the indicator of the shortest cycle’s properties). As the search proceeds, the blue line would move toward a possible ideal cycle (i.e., the green dot), with the same slope, causing the smaller hatched trapezoid becomes smaller and smaller. In case of not existing an ideal cycle, the line stops somewhere in the middle of the bigger hatched trapezoid (which means there would be not better solutions after that point). The blue line at that moment would be an indicator for the properties of most reasonable solution(s). Obviously, in case an ideal cycle exists, the blue line would reach the green dot which means the ideal cycle would be the solution.

Figure 2.5 A length-risk plot for showing the area which is searched for a solution
3 Data Description

For this work, the company provided a sample data of a current project at a power plant containing the position of objects, routes and levels in form of 3D coordinates. There were five\(^2\) sets of objects (i.e., sets of 5, 7, 8, 10 and 14 objects) to be connected by different cable cycles, independently. There were also one set of 78 routes as well as one set of 10 levels. The levels are ceiling or floors of a project building (could be seen as horizontal-planar barriers) which limit access to objects from routes. However, for every object it is guaranteed to be accessible at least through one route. In the following, more details of the three types of dataset (i.e., sets of routes, objects and levels) will be given.

3.1 Set of Routes

A route set contains starting and ending points of straight lines in three-dimensional space, where the lines represent the fixed cable routes. The routes or line are not diagonal. Thus, the changes in coordinates of a line are only along one of the \(x\)-, \(y\)- or \(z\)-axes. The routes can be divided into two groups: vertical and horizontal routes. The vertical routes cross different levels of a building, perpendicularly along the \(z\)-axis. The horizontal routes are located between two levels of a building, either along the \(x\)-axis or the \(y\)-axis.

All the routes are joined together and form one whole structure where every route at least crosses one other route. That is, there is always at least one path from one point on a route to another point on any other routes. For clarification, data of a three-route structure is given as an example (see, Table 3.1), coupled with its visualization (see, Figure 3.1). Figure 3.2 is also visualization of the 78-route data set of the mentioned project (i.e., the power plant).

Table 3.1 Example of a three-route structure dataset

<table>
<thead>
<tr>
<th>Route No.</th>
<th>(x_1)</th>
<th>(y_1)</th>
<th>(z_1)</th>
<th>(x_2)</th>
<th>(y_2)</th>
<th>(z_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-8.80</td>
<td>-3.65</td>
<td>32.70</td>
<td>-8.80</td>
<td>-3.65</td>
<td>48.50</td>
</tr>
<tr>
<td>2</td>
<td>-8.80</td>
<td>-20.20</td>
<td>36.70</td>
<td>-8.80</td>
<td>37.00</td>
<td>36.70</td>
</tr>
<tr>
<td>3</td>
<td>-9.90</td>
<td>-3.65</td>
<td>33.90</td>
<td>5.10</td>
<td>-3.65</td>
<td>33.90</td>
</tr>
</tbody>
</table>

\(^2\) There were more than five sets of objects to be connected in the project, but for this work’s experiments five sets were provided.
3.2 Set of Objects

An object set contains the positions of the objects where a position is specified with three values \((x, y\ and\ z)\) indicating an object coordinate in three-dimensional space. An object might be located on a route, or near a route. An object located on a route is obviously accessible via that route. Whereas, an object near a route is accessible via the route if the following two conditions are true:
1. There must be no obstacle (e.g., a level) between the object and the access point on the route
2. The route must be the nearest route to the object

One requirement for checking the above conditions is finding the access points. An access point or an entry point is the nearest point on a route to an object. The distance between a point on a route and an object is measured by the 3D Manhattan distance metric. A Manhattan or taxicab distance between two points is measured along axes at right angles [15]. The points can be in any dimension. In other words, the Manhattan distance between two vectors $u = (u_1, u_2, ..., u_n)$ and $v = (v_1, v_2, ..., v_n)$ in $n$-dimensional space is the sum of the distances in each dimension [16]. That is:

$$d(u, v) = \sum_{i=1}^{n} |u_i - v_i|$$

For finding the nearest route to an object (to check the second condition), the distance between the object and every route is calculated. The distance, in fact, is the distance between the object and its access point to a route, which is called access distance. Among all of the routes, a route with the shortest access distance to an object is the nearest route to that object.

Since the only way to access to an object would be through its access point, objects are represented by their access points in the search algorithms in this work. That representation only affects the length of a solution but not the main path. Therefore to fix it and decide the real length of a solution, the fixed access distances are added up to the length of a solution at the end of the search. More details of the access points and the nearest routes will be presented further in Section 5.2.

### 3.3 Set of Levels

A level set contains the positions of the levels in a building. In reality, a level might not be in a rectangular shape. That makes the data representation and the computation cumbersome. For this, a level could be partitioned to some rectangular sublevels (see, Figure 3.3).

Figure 3.3 Partitioning a non-rectangular level into three rectangular sublevels
After the partitioning, the rectangular sublevels can be indicated by the coordinates of any two opposite corners of them, as illustrated in Figure 3.4.

![Figure 3.4 Indicating a rectangular sublevel by coordinates of two opposite corners](image)

Thus, in fact, the set of levels contains the mentioned two coordinates of rectangular levels and sublevels.
4 Problem Formulation

As mentioned before, the raw data is in the form of coordinates. However, it is required to be transformed into the form of a graph. The reason and method of the transformation will be discussed later in the next chapter, but for now, it is required to define the elements of the graph for formulating the problem.

First, let us define the nodes of the graph. Indeed, the nodes are in two types with respect to their nature:

1. **Required nodes** representing the objects
2. **Optional nodes** representing the intersections and the end points of the routes

Since a required node represents an object it must be included in a cycle solution in any condition, while an optional node could be missed in a cycle when the inclusion of the node only increases the cost (i.e., length or risk).

Now, let us define the edges. An edge is a route segment between two nodes of any type. Therefore, a route is segmented into weighted edges based on objects and intersections it contains. The weight of an edge is a geometrical distance between the two nodes linked by the edge. For clarification, consider Figure 4.1 which is an illustration of a simple case transformation. In the figure, a two-route structure and one object (Figure 4.1a) has been transformed into a graph (Figure 4.1b).

![Figure 4.1 Illustration of (a) routes and object as well as (b) the corresponding graph](image)

---

3 One of the required nodes is a primary node (representing the primary object described in 1)
4 Since an edge is a straight line along only one of the x-, y- or z-axes, the distance could be either Manhattan or Euclidean
In the example, since Route 1 is segmented into Edge 1, Edge 2 and Edge 3, its length would be equal to a sum of the weight of the edges. Similarly, the length of Route 2 would be equal to a sum of the weight of Edge 4 and Edge 5. That is:

\[
\begin{align*}
\text{length(Route 1)} &= \text{weight(Edge 1)} + \text{weight(Edge 2)} + \text{weight(Edge 3)} \quad (4.1) \\
\text{length(Route 2)} &= \text{weight(Edge 4)} + \text{weight(Edge 5)} \quad (4.2)
\end{align*}
\]

Here, a feasible or a valid cycle is defined. Given a graph \( G \) with a set of edges \( E \) and a set of nodes \( N \) (of which one is primary, some required and some optional nodes). A valid cycle \( VC \) is a path that starts from the primary node and visits each of the required nodes at least once and returns to the primary node at the end, without visiting any edge more than twice. Such a cycle may contain any of the optional nodes, once or more, but not necessarily all of them. By this definition, the shortest cycle in a graph could precisely be defined as follows. The shortest cycle is a valid cycle which minimizes the length. To formulate it, first, a length function for \( VC \) is defined by Eq. 4.3, where the length of an edge \( e \) is denoted by \( l_e \).

\[
\text{length}(VC) = \sum_{e \in VC} l_e \quad (4.3)
\]

By means of that, \( VC_i \) would be the shortest cycle if:

\[
\text{length}(VC_i) \leq \text{length}(VC_j), \quad \forall j: j \neq i \quad (4.4)
\]

Now to formulate the risk, first, a risky edge in \( VC \) is defined as below:

if \( e \in VC \) with \( \text{occurrence}(e) = 2, \) \( e \) would be a risky edge

Also, a lose function for \( VC \) is defined such that \( \text{lose}(e_{\text{risky}}) \) returns the number of required nodes (i.e., the objects) which would be lost if \( e_{\text{risky}} \) is disconnected. By means of the length and the lose functions, the risk function is defined below:

\[
\text{risk}(VC) = \sum_{e_{\text{risky}} \in RE} \text{lose}(e_{\text{risky}}) \text{length}(e_{\text{risky}}) \quad (4.5)
\]

, where \( RE \) is the set of all risky edges in \( VC \) such that \( RE \subset VC \).

Note that the unit of a measured risk is \( mO \), where \( m \) represents meter and \( O \) object.

Now, as an example, the risk for the valid cycle in Figure 4.2 is calculated. The cycle links a primary node to four required nodes, containing three risky edges. First, the number of lost required nodes, when every risky edge is cut, is decided separately as follows:

\[
\begin{align*}
\text{loss}_{e_{\text{risky}1}} &= \text{lose}(e_{\text{risky}1}) = 1 \\
\text{loss}_{e_{\text{risky}2}} &= \text{lose}(e_{\text{risky}2}) = 2
\end{align*}
\]
loss_{erisky3} = \text{lose}(e_{risky3}) = 1

Now the risk is calculated by using Eq. 4.5 as follows:

\[
\text{risk}(VC_{fig.4.6}) = 1 \times \text{length}(e_{risky1}) + 2 \times \text{length}(e_{risky2}) + 1 \times \text{length}(e_{risky3})
\]

Figure 4.2 Example of a valid cycle with three risky edges

As mentioned earlier, since a shortest cycle is the most desired solution from financial point of view, the company is not interested in any solution with a length longer than certain amount higher than the length of the shortest cycle. That amount is determined by setting a parameter, called acceptable extra length (mentioned in Section 2.2 – denoted by \(l_{\text{extra}}\)), for every instance of the problem. In other words, the length of a shortest cycle (denoted by \(l_{lb}\), since it is a lower bound for the length) is added up with the acceptable extra length to result in an upper bound (denoted by \(l_{ub}\)) for the length of a reasonable solution. In addition, the risk of a shortest cycle would be an upper bound (denoted by \(r_{ub}\)) for the risk, because any cycle with a risk greater than a shortest cycle’s risk cannot be a reasonable solution (since it would also be longer than the shortest cycle). The mentioned upper and lower bounds could be represented as below:

\[
l_{lb} = \text{length}(SHC) \tag{4.6}
\]

\[
l_{up} = l_{lb} + l_{\text{extra}} \tag{4.7}
\]

\[
r_{ub} = \text{risk}(SHC) \tag{4.8}
\]

, where \(SHC\) is a shortest valid cycle.
Since it is required to have evaluation criteria for deciding whether a valid cycle is reasonable, ratios (based on the risk and length factors) are defined to indicate deviation of a candidate cycle from a shortest cycle (which is economically optimal) and rank all candidates. So, a length ratio $l_{rc}$, a risk ratio $rr_{rc}$ and an overall ratio $or_{rc}$ for a valid cycle $VC$ are defined as below:

$$l_{rc} = \frac{l_{vc} - l_{lb}}{l_{lb}}, \text{ where } l_{lb} \leq l_{vc} \leq l_{ub} \rightarrow 0 \leq l_{rc} \leq \frac{l_{ub} - l_{lb}}{l_{lb}}$$ \hspace{1cm} (4.9)

$$rr_{rc} = \frac{r_{vc} - r_{ub}}{r_{ub}}, \text{ where } 0 \leq r_{vc} \leq r_{ub} \rightarrow -1 \leq rr_{vc} \leq 0$$ \hspace{1cm} (4.10)

$$or_{rc} = l_{rc} + rr_{rc}, \text{ where } -1 \leq or_{rc} \leq \frac{l_{ub} - l_{lb}}{l_{lb}}$$ \hspace{1cm} (4.11)

Also, a function based on Eq. 4.11 is defined to return the overall ratio of a cycle such that $or(VC)$ returns the overall ratio of the valid cycle $VC$. By means of that a valid cycle $VC_i$ would be most reasonable if:

$$or(VC_i) \leq or(VC_j), \text{ for } \forall j: j \neq i$$ \hspace{1cm} (4.12)

That is to say, a valid cycle with the minimum overall ratio would be the most reasonable cycle.
5 Data Transformation

In the method proposed in this work, other than constructing a graph, some information and complementary data are extracted and added to the graph as well. Figure 5.1 gives an overview of the data and information structure needed for further process.

To extract such data and information from the raw data, a data transformation and preprocessing are done. All the coordinates indicated in the figure are already available as the raw data. The route orientations, access points and construction of the pure graph (without the signs) will be elaborated in the following subsections. The procedure of adding signs to the graph will be discussed in Subsection 6.3.3.

5.1 Route Orientation

As stated in Section 3.1, the routes can only be along one of the x-, y- or z-axes. Therefore the routes are in three types with respect to their orientations. The orientations can be indicated by numerical or alphabetical tags (e.g., 1, 2 and 3 indicating “along x”, “along y” and “along
The orientation of a route is determined by looking at the change in the head and tail coordinates of the route. Since the change is always in one dimension, subtracting one coordinate from another will result in only one non-zero value out of the three. From that, it would be easy to decide the orientation of the route and tag the route.

5.2 Object Access Point

The access points and the nearest route to an object were introduced in Section 3.2. Below, the process of finding an accessible nearest route and the position of an access point are explained.

First of all let us see how the nearest point on a route to an object could be determined. Given an object \( obj \) with coordinate \( \text{obj}_\text{pos} = (x_{obj}, y_{obj}, z_{obj}) \), and a route \( rt \) with head and tail’s coordinates \( \text{rt}_{\text{head}} = (x_{rth}, y_{rth}, z_{rth}) \) and \( \text{rt}_{\text{tail}} = (x_{rtt}, y_{rtt}, z_{rtt}) \). The nearest point \( \text{np}_\text{pos} = (x_{np}, y_{np}, z_{np}) \) on the route \( rt \) to the object \( obj \) is determined as below:

\[
x_{np} = \text{median}((x_{obj}, x_{rth}, x_{rtt})) \quad (5.1)
\]
\[
y_{np} = \text{median}((y_{obj}, y_{rth}, y_{rtt})) \quad (5.2)
\]
\[
z_{np} = \text{median}((z_{obj}, z_{rth}, z_{rtt})) \quad (5.3)
\]

After finding the position of a nearest point, the accessibility of the point from the object should be checked. As mentioned earlier, a point is accessible from an object if there is no level in between. A level \( \text{lev} \) with two opposite corners’ coordinates \( \text{lev}_{c1} = (x_{lc1}, y_{lc1}, z_{lc1}) \) and \( \text{lev}_{c2} = (x_{lc2}, y_{lc2}, z_{lc2}) \) would be between an object \( obj \) and a nearest point \( np \), if all the conditions in below are satisfied:

\[
\min(x_{lc1}, x_{lc2}) < \{x_{obj}, x_{np}\} < \max(x_{lc1}, x_{lc2}) \quad (5.4)
\]
\[
\min(y_{lc1}, y_{lc2}) < \{y_{obj}, y_{np}\} < \max(y_{lc1}, y_{lc2}) \quad (5.5)
\]
\[
\min(z_{obj}, z_{np}) < z_{lc1} < \max(z_{obj}, z_{np}), \quad \text{where } z_{lc1} = z_{lc2} \quad (5.6)
\]

As soon as all the accessible routes from an object \( obj \) are detected, the distance between the object and every accessible nearest point \( np \) is obtained by a special case of Eq. 3.1 as below:

\[
d(\text{obj}_\text{pos}, \text{np}_\text{pos}) = |x_{obj} - x_{np}| + |y_{obj} - y_{np}| + |z_{obj} - z_{np}| \quad (5.7)
\]
It is also required to know whether or not a nearest point on an accessible route is the nearest one among all other nearest points on other accessible routes. An accessible nearest point with the shortest distance to an object would be the access point of the object, and the route to which the access point belongs would be the access route of the object.

### 5.3 Graph Construction

There are many reasons to convert the data into the form of a graph and then have it signed. First and important reason is the existence of many algorithms for solving problems that are modelled in the form of a graph [17]. Some of those algorithms, or at least ideas inspired from them, are used in this work to solve the problem; one example is the Dijkstra’s algorithm explained in Appendix C which is employed in finding shortest paths between nodes.

Due to the route intersections and the position of an access point, which is mostly between the two bottoms of a route, a solution cycle may include only a piece of a route in its path. To take those candidates into account, it would be easier to work with a graph in which the routes are segmented into edges. Also, since a graph provides the topology of the routes and objects, recurring checking the connectivity of the points and the distance between them would be easier and the computational burden considerably would reduce. Furthermore, by the method proposed in this work, one can sign every node of a graph to have a guided search. By means of the signs and knowing the current and previous states of a search, further exploration of wrong paths could be prohibited.

To construct a graph, the initial step is finding the intersections between the routes. One requirement for two routes to cross each other is that both must be in a same plane, parallel with one of the \(xy\)-, \(xz\)- or \(yz\)-planes. That is, for two routes \(rt_1\) and \(rt_2\) with the following coordinates:

\[
rt_{1\text{head}} = (x_{rt1h}, y_{rt1h}, z_{rt1h})
\]
\[
rt_{1\text{tail}} = (x_{rt1t}, y_{rt1t}, z_{rt1t})
\]
\[
rt_{2\text{head}} = (x_{rt2h}, y_{rt2h}, z_{rt2h})
\]
\[
rt_{2\text{tail}} = (x_{rt2t}, y_{rt2t}, z_{rt2t})
\]

one of the conditions in below must be satisfied:

\[
x_{rt1h} = x_{rt1t} = x_{rt2h} = x_{rt2t}
\]

\[
y_{rt1h} = y_{rt1t} = y_{rt2h} = y_{rt2t}
\]
Two routes in the same plane must also have a different orientation to cross each other. Moreover, based on the orientation of the routes, the coordinates must be checked mutually. However, there is a tricky way to simply check if two routes cross each other. In that way it is also easier to find the intersection point, if any. To this end, first, the coordinates of the two routes are combine together and sort (for every dimension separately), and keep the result in arrays as follows:

\[
X = \text{sort}((x_{rt1h}, x_{rt1t}, x_{rt2h}, x_{rt2t})) \tag{5.11}
\]

\[
Y = \text{sort}((y_{rt1h}, y_{rt1t}, y_{rt2h}, y_{rt2t})) \tag{5.12}
\]

\[
Z = \text{sort}((z_{rt1h}, z_{rt1t}, z_{rt2h}, z_{rt2t})) \tag{5.13}
\]

Next, all the three conditions in below must be satisfied to have an intersection:

\[
\]

\[
\]

\[
\]

So, the intersection point \( \text{ints}_{pos} = (x_{ints}, y_{ints}, z_{ints}) \) could be easily determined by:

\[
x_{ints} = X[2] \tag{5.17}
\]

\[
y_{ints} = Y[2] \tag{5.18}
\]

\[
z_{ints} = Z[2] \tag{5.19}
\]

As soon as the list of the intersection coordinates is provided, the coordinates of the access points and the routes\(^5\) is added to the list to build up a unique list for the coordinates of all nodes. The rest of the procedure could be followed in Figure 5.2.

\[^5\text{The two end points of the routes, each as one node}\]
procedure buildGraph(routes, objects)
    nodesCoord ← combine intersections', access points’ ... and routes’ coordinates as one list
    nbNodes ← number of nodes  // length of nodesCoord
    nbObjects ← number of objects
    edgeCoord, edgeParentRoute, edgeLinkingNodes, ...
        edgeLength: empty lists
    nodeCorrespondingObj: array[1..nbNodes]
    connectivityMap: matrix[1..nbNodes][1..nbNodes]
    distanceMatrix: matrix[1..nbNodes][1..nbNodes]
    initialize nodeCorrespondingObj with zero
    copy each node label into corresponding element ...
        of nodeCorrespondingObj
    initialize connectivityMap with zero
    initialize the main diagonal of distanceMatrix ...
        with zero and the rest with infinity
    for every route rt in routes do
        nodesOnCurrentRoute: an empty list
        for every node nd in nodesCoord
            if nd is on rt
                add nd to nodesOnCurrentRoute
        sort nodesOnCurrentRoute based on the coordinates
        for i = 1 to (number of elements in ...
            nodesOnCurrentRoute) - 1 do
                nodeSrc ← nodesOnCurrentRoute[i]
                nodeDst ← nodesOnCurrentRoute[i + 1]
                add coordinates(nodeSrc, nodeDst) to edgeCoord
                add rt to edgeParentRoute
                add (nodeSrc, nodeDst) to edgeLinkingNodes
                add dist(nodeSrc, nodeDst) to edgeLength ...
                    and distanceMatrix(nodeSrc, nodeDst)
                add edge label to ...
                    connectivityMap(nodeSrc, nodeDst)

Figure 5.2 Procedure of building a graph
6 Methods

Before coming up with the final method for solving the problem, attempts were made on other methods which failed. This chapter explains those methods and the reasons of their failures, before proposing the final method. The first following section describes the issues for developing approximate algorithms. Next section explains another unsuccessful method and its issues. Finally, the last section proposes the final method.

6.1 Issues for Developing Approximate Algorithms

Generally the search algorithms are into two types:

1. *Exact algorithms* which provide an exact or optimal solution.
2. *Approximate algorithms* which provide a suboptimal or near the optimal solution.

Often an exact search begins with a partial candidate (i.e., a node) and continues with candidates which are derived from the initial candidate (i.e., the initial node is expanded). That search ends when whole the search-space\(^6\) is explored. On the contrary, an approximate search begins with an initial complete candidate, continues with other candidates which are generated based on the search strategy, and ends by reaching stopping criteria. In both types of the search, every new candidate is evaluated\(^7\) by a function for a comparison or other purpose. The evaluation function gives the cost of a candidate that could be based on length, risk or other factors.

One may think of adapting a search algorithm which works for a particular problem with a specific evaluation factor, for the same problem but with different factor(s), by simply replacing the evaluation function. This is possible if the search-spaces are the same. That is, the search algorithm which is going to be adapted should be able to produce all the candidates needed for the new problem. However, due to reasons, it is not always possible to have such a matching between the search-spaces.

Let us consider that issue in terms of the current problem. As mentioned in Section 1.1, there are many problems (e.g., the problem of order-picking in a warehouse) which are same as the current problem, if the only goal is minimization of the total length of the cable. As a method for solving such problems, it has been suggested to transform the problem into an instance of the TSP and solve it by one of the available approximate algorithms for solving the TSP. That transformation is because of two issues which are explained in the following.

---

\(^6\) Sometime by some techniques the size of a search-space is reduced without losing any potential candidate.

\(^7\) Due to the way a candidate is generated sometimes it is required to validate a candidate as well.
Since in the TSP one is looking for the shortest Hamiltonian cycle\(^8\), the graph of the problem should necessarily be a Hamiltonian graph\(^9\). However, the graph of the current problem is not guaranteed to be Hamiltonian. Other than this issue, all the nodes of the graph are not required to be included in the cycle (it is only required to connect the objects, not necessarily the route intersections or other points as nodes). Thus, an alternative is to transform the graph into a Hamiltonian graph and eliminate all optional nodes. This could be done by identifying the shortest path between every pair of the required nodes and establishing a new graph with only those nodes where edges represent the shortest paths. Let us consider the example in Figure 6.1 which shows a transformation from an original graph of the problem into a Hamiltonian graph based on the shortest paths. In Figure 6.1a which shows the original graph, the edges represent the route segments and the weights beside them indicate the distance between the nodes. In Figure 6.1b which is the transformed graph, the edges represent the shortest path between the required nodes. Note that the optional nodes (the blue nodes) which represent intersections and a dead-end were eliminated in the transformed graph, because it is assumed the shortest path between two required nodes is unique\(^{10}\).

![Figure 6.1 Transformation of (a) an original graph of the problem to (b) a Hamiltonian graph based on the shortest path between the required nodes](image)

As soon as the original graph is transformed into a Hamiltonian graph, an approximate algorithm like simulated annealing could be employed to find a near the shortest cycle which links all the required nodes. Although the explained method is useful in solving the mentioned problem, it could not be applied for the current problem. The reason is an issue

---

\(^8\) A Hamiltonian cycle is a path through a graph that starts and ends at the same node and visits every other node exactly once [32].

\(^9\) A graph that contains at least one Hamiltonian cycle is called a Hamiltonian graph.

\(^{10}\) In finding the shortest cycle, even if there are more shortest paths between a pair of required nodes, each one could be chosen as a shortest path without any effect on the final solution.
which arises when the original graph is transformed. The search-space (when the risk is taken into account) with a transformed graph could not be produced completely and some potential candidates might be missed. That is, some paths which are longer than the shortest paths (between the required nodes) would not be considered anymore when a graph is transformed. Those paths might be used when a longer but more reliable solution is required.

To clarify that, let us back to the example in Figure 6.1. If one tries to find the shortest cycle for connecting the required nodes A, B and C in the original graph, the solution would be the sequence A-e-B-e-C-e (representing the cycle by sequence of nodes). That solution in term of the transformed graph would be the sequence A-B-C. Now, if one tries to find the most reliable cycle for connecting the three required node, the solution would be the sequence A-e-B-f-g-h-C-e. That solution could not be produced by the transformed graph in any way.

Other way one might think to use an approximate algorithm for solving the problem could be working with the original graph to avoid missing any potential candidate. In that way, the path of a cycle might be represented by a complete sequence of nodes including the required and optional nodes. Normally at the beginning of a search by an approximate algorithm, a random sequence is generated as an initial candidate. As the search proceeds, new candidates are produced by a neighbourhood function which usually permutes the nodes of a current candidate, randomly. However, producing a new candidate in that way could not be done due to issues with the sequences based on the original graph. First issue is that, in those sequences, the number of nodes (i.e., the length of sequence) is not fixed and predictable (unlike the sequences in the TSP). Nevertheless, the number of nodes “itself” in not the problem here, because it is an integer between \( n_r \) and \( 2(n_t - 1) \), where \( n_r \) is the number of required nodes and \( n_t \) the total number of nodes. That could be decided by randomly picking an integer, in the mentioned range, before generating a new candidate. The problem arises when inclusion of a node in a sequence is randomly decided (remember it is allowed to include a node several times). Obviously by such a node selection and arrangement, many of the produced sequences would not be a cycle at all. For the same reasons, one cannot build up a neighbourhood function to generate a new candidate based on a current candidate. This means, it is very probable that a new sequence which is produced based on an input sequence (by adding, removing or permuting some nodes) to not be a cycle any more. So, every time, many sequences have to be produced to reach a new cycle. Moreover, if one is lucky and can generate a new cycle from a current cycle, the length and risk of the new one could be very different in comparison with the current one. So in that case, the new cycle would not be a neighbour for the current one. Consequently, a search based on such reproductions would be nothing but a blind search on which there is no control.

Although the problem could not be solved by merely using an approximate algorithm, the advantage of such an algorithm will be taken in part of the final method. In the proposed method which is kind of an exact search, a simulated annealing algorithm will be employed at the beginning of the search for finding a near the shortest cycle. The properties of that cycle
would be used in setting lower and upper bounds for reasonable solutions. That will help reduce the search-space for an exact search. More details will be given in Section 6.3.

### 6.2 A Method Based on Finding Perfect Cycles

This section discusses another method which was developed for solving the problem, though it failed for some cases. The idea came up based on the fact that a perfect cycle (see, Section 2.1) which connects all the objects has no risk of failure in case of a single link cut. It was already mentioned that such a cycle might be expensive (i.e., long) or not existing, but there might be perfect cycles which connect a subset of the objects. Among those cycles, some of them which maximize the number of connected objects could be selected as initial cycles to be expanded to include the remaining objects. Nodes and edges which are included in an initial cycle would not be considered anymore in further expansion of that cycle. Afterward, new searches among the remaining nodes and edges would be done for finding other possible perfect cycles which maximize the number of connected objects. Then, the new cycles would be merged (unified) with the initial cycles in all the possible ways to form new cycles. This would be done again and again until either all the objects become connected or there remain objects which cannot form a perfect cycle. In the second case, every remained object would be joined with its corresponding expanded cycle in all the possible ways to form new cycles. At the end, there would be many complete cycles among which one with the minimum cost would be the best one. Figure 6.2 depicts an example of the above procedure. In the figure, drawings which include dashed lines indicate nodes and edges which should be considered in a next search. Also, drawings which include purple lines indicate found or merged cycles. Every rejected and accepted cycle has been signed with a cross and check mark, respectively.

Although the explained approach works well in many cases, in some specific cases two issues were detected. The first issue is that in some cases one can find a better solution with a little more risk and a very shorter length than a solution provided by the mentioned approach. The second issue arises for a graph in which there are many maximizing perfect cycles which link only few objects. The issue is that the unification process in those cases is very time-consuming, due to the process of checking all the possible ways of merging perfect cycles or remaining objects. However, the mentioned approach works very well if the goal is only finding the most reliable cycle.
This section, first, explains a depth-first search and presents some techniques for making a search-space smaller. Next, an overview of the proposed method is given, and then every step of the method is considered in details.

6.3 Proposed Method

A depth-first search is kind of a brute-force search for traversing a tree to reach an exact solution or a global optimum. The search starts from the root of a tree and moves deeply throughout the tree. At every step, if the current child node is a leaf, the search should be continued from the first previous node which has at least one unvisited branch. Then, one of the unvisited branches should be traversed until reaching another leaf. This going forth and backtracking is carried on until finding a solution or traversing the entire tree. Figure 6.3 shows an example of a DFS search in a tree, where node labels indicate the order of node traversal.

To avoid an infinite loop while doing a DFS in an undirected graph, it is necessary to keep track of the visited nodes in every tuple by using a memory. However, in some problems (like the current one) it is required to visit a node more than once (but of course not infinitely). Also, due to the nature of DFS (or generally brute-force search), it is only applicable for trees
or graphs with few nodes. However, sometimes it works for particular big problems by using some techniques.

One technique is an enhancement of the backtracking mechanism of DFS. This is known as the backtracking algorithm\textsuperscript{11}. The backtracking algorithm exploits a validation function in order to avoid further expansion of partial candidates which end in an infeasible solution. This early stopping of the traversal of a fruitless branch makes it possible to save much time in many problems.

Another technique is to lead the search to the solution space and that is known as branch-and-bound. The idea is similar to backtracking, but branch-and-bound takes a heuristic approach to detect and traverse more gainful branches. A branch-and-bound algorithm compares the cost of the best solution found so far (i.e., an upper bound in case of minimization) with an estimated minimum cost of complete solutions reachable from the current state. If the estimated cost exceeds the upper bound, progressive movement from the current state should be prohibited. One can implement BB with a ranking of the states (based on heuristic evaluation) to prioritize expansion of the most promising nodes, and one can only prune the branches ending to high cost solution \cite{18}; the first is similar to best-first search. Despite of the advantages of the prioritization, it may not help for a faster search than a simple BB, in some problems. One reason is the usage of a priority queue that requires additional computation \cite{19}.

Due to the similarities of backtracking and branch-and-bound, one can combine the two mechanisms in a single algorithm to take advantages of both. The algorithm could be implemented recursively or non-recursively using stack (or queue when the search strategy is BFS\textsuperscript{12}).

\begin{figure}
\centering
\includegraphics[width=0.3\textwidth]{fig6.png}
\caption{Example of a depth-first search in a tree}
\end{figure}

\textsuperscript{11} Backtracking algorithm should not be confused with the naive backtracking DFS algorithm performs, since DFS backtracks only on leaves but backtracking algorithm do it in more sophisticated manner (can be done earlier than reaching a leaf).

\textsuperscript{12} Breadth-first search (BFS) is similar to DFS, but as the name encourages before going to a next level of a tree, all nodes in the current level should be checked sequentially.
6.3.2 Overview of the Method

The search strategy of the proposed method would be a DFS using the two techniques described in the earlier subsection. To do the backtracking, a powerful validation function will be employed (will be explained in Subsection 6.3.5). Also, as stated before, a simulated annealing algorithm will be used for finding a near the shortest cycle and deciding the upper and lower bounds defined by Eq. 4.6, 4.7 and 4.8. To do the pruning, the risk of every partial solution will be evaluated and the minimum length of a final solution will be estimated as well (see the details of that estimation in Subsection 6.3.6). Furthermore, to reduce the computational burden of the validation and evaluation processes, some signs will be added to the original graph in order to avoid generating many invalid or infeasible candidates.

For a better understanding of the method, an overview of the steps is provided in below:

1. The data is transformed into the form of a graph (discussed in Chapter 5).
2. Signs and conditional directions are added to the original graph (see, Subsection 6.3.3)
3. The shortest distance and path between the required nodes (i.e. objects) is decided by Dijkstra’s algorithm (see, Subsection 6.3.4 and Appendix C) and an adjacency matrix is formed for that.
4. A near the shortest cycle is found by passing the adjacency matrix (formed in step 3) to a simulated annealing algorithm (see, Subsection 6.3.4 and Appendix B).
5. Lower and upper bounds for risk and length are calculated based on the properties of the near the shortest cycle found in step 4 (by using Eq. 4.6, 4.7 and 4.8).
6. A depth-first search with the backtracking and branch-and-bound approaches (discussed in Subsection 6.3.1) is started for finding a final solution. There are also some remarks about the search as follows:
   - In exploration of the graph the signs and conditional directions (see, Subsection 6.3.3) are considered.
   - Every invalid partial solution is rejected by a validation function (see, Subsection 6.3.5).
   - The partial and complete solutions are evaluated based on their overall ratio (calculated by using Eq. 4.11).
   - Every partial and complete solutions with a length and risk more than the length and risk upper bounds is rejected (pruned)
   - The length for every partial solution is an estimation of minimum length of completing the partial solution (see, Subsection 6.3.6)
   - The risk for every partial solution is its exact current risk.
   - Every complete and valid cycle with the minimum overall ratio is accepted as the most reasonable solution.

(For an implementation purpose, the details of step 6 are available in Appendix D)
6.3.3 Adding Signs to the Graph

The explained procedure in Chapter 5 for building a graph provides an undirected-weighted graph. However, for the sake of a guided search (which is one of characteristics of this work), it is required to add signs to the nodes. A signed graph should not be confused with a directed graph. In a directed graph, directions depend only on a current state, whereas in a signed graph they depend on both the current and previous states. Additionally, in a signed graph, one can be informed about the nodes on the ways (intermediate nodes) from a current node to next branchy or dead-end nodes (these nodes will be explained further).

The signs are categorized as follows:

1. Signs indicating the nodes on the ways to next branchy or dead-end nodes
2. Signs indicating conditional directions

To have the first type of the signs for a graph, one does not need to have prior knowledge about the problem. Type 1 signs for two graphs with identical topology are the same. On the contrary, it is not possible to determine type 2 signs for a graph, when the problem is not clear. This means, one should know what node sequences in the search of a graph end in invalid solutions. Two identical graphs (i.e., same topology) of two different problems would have similar type 1 signs but possibly different type 2 signs. To have signs of the second type for a graph, type 1 signs should be decided beforehand. Before going through the details of the signs, some general definitions are provided below.

The degree of a node is the number of edges incident to the node with self-loops counted twice [20]. The degree of a node \( n \) is denoted by \( \text{deg}(n) \). In this work, nodes are categorized into three types with respect to their degree, as below:

1. Death-end node\(^{13} \) is a node with degree 1 or \( \text{deg}(n) = 1 \)
2. Intermediate node is a node with degree 2 or \( \text{deg}(n) = 2 \)
3. Branchy node is a node with degree more than two or \( \text{deg}(n) > 2 \)

The number of signs of each type for every node would be equal to the degree of the node. This means, for instance, a node of degree 3 has 3 signs of type 1 and 3 signs of type 2 (i.e., two signs for every edge incident).

Now, let us take a closer look at the first type of the signs. This can be observed on a graph that via every edge incident to a node, one can reach a dead-end or a branchy node either immediately or after passing \( n \) intermediate nodes, where \( n \) is a positive finite integer. A sign

\(^{13}\)In the graph theory, a death-end node is called a leaf.
of type 1 indicates all those intermediate nodes (if any) as well as those dead-end or branchy nodes for every edge incident to a node. Figure 6.4 is an illustrated example of type 1 signs in a simple five-node graph.

Also, the procedure of signing an undirected graph with type 1 signs comes in Figure 6.5 for an implementation purpose.

```
procedure type1sign(graph)
    for every branchy and dead-end node sN in graph do
        for every adjacent node adjN to sN do
            currentNode ← adjN
            nodesOnTheWay: empty list
            while currentNode is intermediate node do
                if sN is dead-end node
                    sign ← {reverse(nodesOnTheWay), ...
                            sN, ‘-1’}
                else
                    sign ← {reverse(nodesOnTheWay), sN}
                assign sign to currentNode for the ...
                way toward sN
                add currentNode to nodesOnTheWay
            currentNode ← next unvisited adjacent ...
                to currentNode
            if currentNode is dead-end node
                sign ← {nodesOnTheWay, currentNode, ‘-1’}
            else
                sign ← {nodesOnTheWay, currentNode}
            assign sign to sN for the way toward ...
        currentNode
```

Figure 6.5 Procedure of signing an undirected graph with type 1 signs

14 In case of a dead-end node, a “-1” is added to the end of the sign tuple as well to make a distinction between the two cases (i.e., a dead-end and branchy node)
Type 2 signs are kind of directions to guide further transitions between the states depending on a current partial solution. In an undirected and unsigned graph, it is always possible to have transition between two adjacent nodes. That kind of freedom may lead a search to an infinite loop, infeasible solutions or non-reasonable solutions. One may use a validation plus an evaluation functions to solve rather this problem. Those functions are used to avoid going into the wrong states, but sometimes they are computationally heavy. Therefore, their usage for every transition in a brute-force search may not be a good idea.

One can lead a search to most feasible and possible-to-be-optimal solutions by defining common directions which are used in the graph theory. By means of them, many recurrent and time-consuming tests for a validation or/and an evaluation could be avoided. However, it would be possible to miss some of potential candidates by ordinary directions. One alternative could be defining conditional directions which depend on the current and previous states of a search.

The conditional directions are not a replacement for a validation or an evaluation functions. They are only used to make the functions simpler for faster computations. Unlike the common directions, the conditional directions are defined on the nodes, not on the links. As mentioned before, defining the conditional directions varies from one problem to another and needs a clear understanding of a problem.

Now, let us consider the undesired cases which are prevented by those directions. In general, the conditional directions prevent wrong returns at some specific nodes and under specific conditions. A “return” is defined as a transition from a current state to a previously visited state. In other words, a return is two-time occurrence of an edge in a cycle, consecutively. In case of representing a cycle by a node sequence, a return would be locating a node between two identical nodes. For instance, in the node sequence \((n_1, n_3, n_5, n_7, n_5, n_2)\) the node \(n_7\) is a return point to the node \(n_5\). Generally, the returns which are prevented by using the conditional directions are categorized into the following three types:

1. Any return at an optional node
2. Particular returns at required nodes on a branch\(^{15}\)
3. Particular returns at required nodes on a dead-end path\(^{16}\)

In the following, the details of the mentioned types of returns will be given and it will be shown that a solution including any of them could not be reasonable in any way.

\(^{15}\) A branch is a path between two branchy nodes without any other branchy node in between

\(^{16}\) A dead-end path is a path between a branchy node and a dead-end node without any other branchy node in between
The first type of returns, which is a simple return at an optional node, does not provide any benefit for a cycle. This means, such a return only increases the length of a cycle without linking any new required node.

Removing such a return from a cycle will result in a new cycle. The new cycle would have a shorter length and the same risk compared to the cycle including the return. To prove it, assume a valid cycle \( VC_1 \) containing one return at an optional node and a new valid cycle \( VC_2 \) which will be produced if the return is removed from \( VC_1 \). It will be shown that \( VC_2 \) is shorter than \( VC_1 \) with the same risk. Assume \( e_{\text{return}} \in VC_1 \) is the edge through which the return occurs. So, \( VC_2 \) does not contain \( e_{\text{return}} \) and \( VC_2 \subseteq VC_1 \). Consequently, it could be concluded that \( \text{length}(VC_2) < \text{length}(VC_1) \). Also, since no required node would be disconnected by removing \( e_{\text{return}} \) from \( VC_1 \), according to the definition of the risk, it could be concluded that \( \text{risk}(VC_2) = \text{risk}(VC_1) \). Thus, it was proved that \( VC_2 \) would be a better solution than \( VC_1 \). Figure 6.6a shows an example of a cycle with a return at an optional node. Removing the edge, through which the return is happening, would result in the cycle showed in Figure 6.6b which is shorter with the same risk.

The second type of returns is the one which happens at some of the required nodes on a branch containing at least two \(^17\) required nodes. Basically, the second type itself could be divided into two groups of unwanted returns. To explain them, first, a best gap on a branch is introduced. A best gap is an edge \( e_{bg} \) of a branch (containing at least two required nodes) that minimizes the sum of the common length \( cl_{bg} \) and the common risk \( cr_{bg} \) of any valid cycle not including \( e_{bg} \), where \( cl_{bg} \) and \( cr_{bg} \) are determined by using Eq. 6.1 and 6.2, respectively.

\[
cl_{bg} = 2 \times \sum_{e \in B - e_{bg}} l_e \tag{6.1}
\]

\[
\text{cr}_{bg} = \sum_{e \in B - e_{bg}} \text{lose}(e) \text{ length}(e) \tag{6.2}
\]

\(^17\) In case any of the two branchy nodes at two end points of the branch is a required node, it would be counted as well.
, where $B$ is the set of all the edges of the branch which is under consideration.

According to the definition of a valid cycle, for a branch $B$ containing at least two required nodes, any valid cycle not passing through $e_{bg} \in B$ has the same $cl_{bg}$ and $cr_{bg}$, because the paths they traverse in $B$ are the same. Although the length and risk of the remaining paths of the cycles (out of the branch) could be different, they are not influential in deciding the best gap (see, Figure 6.7).

![Diagram showing two different valid cycles with the same common paths on a branch](image)

**Figure 6.7 Two different valid cycles with the same common paths on a branch**

To find the best gap of a branch, first, it is required to calculate the sum of the common length and the common risk for every edge of the branch (called common cost or $cc$) after normalization from 0 to 1. After calculating all the common costs, the edge with the minimum $cc$ would be the best gap of that particular branch. The two nodes of the best gap as well as the first and the last required nodes of a branch are crucial in deciding unwanted returns.

The exploration of a branch in a graph could be started from one of its two end nodes. Through each of the nodes the exploration is started, it could be done in two ways. One is to reach the other end node and exit the branch. The other is to return at any node in between and exit through the first explored node. However, as mentioned earlier, there are returns which are not desired. Depending on the end node from which the exploration is started (i.e., direction of movement), the returns are suppressed at:

1. All nodes of the branch before the first node of the best gap
2. All nodes between the first node of the best gap and the last required node of the branch

Figure 6.8 depicts the two groups of the nodes as well as other nodes which play a role in suppressing the second type of the wrong returns. Note that with different movements, the nodes have different roles.

![Diagram](image-url)

**Figure 6.8 Illustration of different types of the nodes on a branch: (a) left to right movement; (b) right to left movement**

Since any kind of a return at an optional node is already inhibited, any new inhibition of such a return in a branch does not make any sense. Therefore, in the following, the suppression of the second type of the wrong returns in a branch only for the required nodes will be considered.

As before, to justify such suppressions, it will be proved that for any solution including a return of the second type, there would be at least one better solution without that return. To this end, without loss of generality, a branch containing three required nodes (see, Figure 6.9) will be considered.
Figure 6.9 Example of a branch with three required nodes: (a) conditional directions; (b) partial valid cycle excluding $e_{bg}$; (c) partial valid cycle excluding $e_{nbg}$

Regarding the branch in Figure 6.9, assume there are two valid cycles $VC_{bg}$ (see, Figure 6.9b) and $VC_{nbg}$ (see, Figure 6.9c) which have the same path out of the branch, but different path in the branch such that $VC_{bg}$ is not including $e_{bg}$ and $VC_{nbg}$ is not including $e_{nbg}$. Moreover, assume $cc_{bg}$, $cc_{nbg}$ and $c_{rem}$ are respectively the common cost of $VC_{bg}$, the common cost of $VC_{nbg}$ and the remaining cost\(^{18}\) of both $VC_{bg}$ and $VC_{nbg}$. The total cost of both cycles is obtained from the following summations:

$$ctotal_{bg} = cc_{bg} + c_{rem}$$

\(^{18}\) As mentioned before, the remaining cost or the out-of-the-branch cost of valid cycles with the same out-of-the-branch path is the same.
\[ c_{total_{nb}} = cc_{nb} + c_{rem} \]

Since according to the definition of the common cost \( cc_{bg} < cc_{nb} \), it could be concluded that:

\[ c_{total_{bg}} < c_{total_{nb}} \]

The signs in Figure 6.9a visualize final conditional directions defined on the nodes of the branch in the given example. Note that other inhibited returns in the figure could be justified in a similar way.

There might be a confusion about the return sign at the first left required node of the branch in Figure 6.9a. According to the preceding justification, that return is undesired; because it leads to a cycle with a higher cost than \( c_{total_{bg}} \) (when the remaining costs are equal). In spite of that, such returns are not suppressed, because in a given graph, a branch might be the only way to go from one of its two ends to another. Hence in such a case, a valid cycle would not be possible without the return at a last required node of the branch.

Finally, the last type of unwanted returns is the one which happens at a required node on a dead-end path under specific conditions. It is easy to see that the only point to enter and exit a dead-end path for an exploration is the branchy node at one ends of the path (i.e., the entrance of the dead-end path). This would cause obtaining invalid solutions when two types of returns are made. One of them is early returns at required nodes on a dead-end path, without connecting other required node(s) ahead. The other one is returns at required nodes during the way of exit from a dead-end path after visiting the last required node on the path. To speed up the search, those types of return are inhibited. An example of an early return in a dead-end path could be observed in Figure 6.10.

![Figure 6.10 Example of an invalid cycle resulted by an early return in a dead-end path](image)

Now, it should be proven that type 3 returns result in invalid solutions. As there are two different kinds of type 3 returns, the proof for each one will be given separately. For the early returns assume \( C_{early} \) is a cycle containing an early return in a dead-end path. The edge
through which the return is made will be denoted by $e_{\text{early}}$. Also, the required node which is missed by that early return will be denoted by $rn_{\text{unlinked}}$. According to the definition of a valid cycle, to make $C_{\text{early}}$ valid, it would be required to include $rn_{\text{unlinked}}$ in the cycle as well. To do that, $e_{\text{early}}$ should be added to $C_{\text{early}}$, because the only way to access $rn_{\text{unlinked}}$ would be via $e_{\text{early}}$. However, due to the early return, $e_{\text{early}}$ is already in $C_{\text{early}}$ twice. Hence, another inclusion of $e_{\text{early}}$ will make the cycle invalid. So, $C_{\text{early}}$ could not become a valid cycle in any way.

It will also be proven that the returns during an exit from a dead-end would be wrong. For this, assume $PC$ is a partial cycle which has already linked all the required nodes on a dead-end path and been valid so far. Also, suppose the last visited node $rn_{\text{current}}$ and edge $e_{\text{current}}$ belong to the dead-end path, where $rn_{\text{current}}$ is not a last required node or dead-end node. In case of a return at $rn_{\text{current}}$, $e_{\text{current}}$ would be included in $PC$ at least for the second time and the direction of the movement would change toward the dead-end node. Consequently, to get out of the dead-end path, another return toward the entrance would be inevitable. That would cause another inclusion of $e_{\text{current}}$ in $PC$ which would make $PC$ invalid. So, it was proved such a return would be wrong.

As soon as the conditional directions are determined, the graph would be ready for the next step which would be finding a near the shortest cycle and deciding the required bounds. The coming subsection describes the relevant process.

### 6.3.4 Finding Near the Shortest Cycle

As stated in Chapter 4, a shortest cycle’s properties (i.e., length and risk) could be used for initiating lower and upper bounds in a search for a most reasonable solution concerning length and risk. However, finding a shortest cycle itself is NP-hard. So as an alternative, an approximate algorithm could be employed to find a near the shortest cycle. Although that type of cycle might be a little longer than the shortest one, the quality of the final solution would not be affected by that (due to using such a cycle only for initiating bounds). The only difference might be a small increment in the computational time of the search.

It is important to know that the “type” of the approximate algorithm which is employed for finding a near the shortest cycle is not crucial. The point is only using an approximate algorithm to make an exact algorithm faster. In this work, a simulated annealing algorithm was adapted, but due to the mentioned reason the configuration of the algorithm is neglected here.

The transformation discussed in Section 6.1 is useful here, because the goal at this step is finding a near the shortest cycle. For that transformation and the simulated annealing, a similar approach to what Farooq [3] did in his work was taken. The difference is more or less
in using different methods for finding the shortest distances and paths between the required
nodes (the difference would be considered further).

In Section 6.1, it was discussed that for finding a near the shortest cycle the original graph
could be transformed into a Hamiltonian graph in order to exploit an approximate algorithm
like simulated annealing. To this end, the shortest path between every pair of required nodes
is needed. As already stated, in a transformed graph, there would be no optional node.
Therefore, a path in a transformed graph does not contain any optional node between the
required nodes. For instance in a simple weighted graph in Figure 6.11, the cycle \((n_1, n_2, n_3, n_4, n_5, n_6, n_7)\) would be equal to cycle \((n_1, n_3, n_6)\) in the transformed graph.

Also, the weight of an edge between every pair of required nodes would be equal to the
length of the shortest path between them in the original graph. A shortest path between a pair
of required nodes could be decided by using any algorithm for solving the shortest path
problem such as Floyd-Warshall or Dijkstra’s. This could even be done by a brute-force
search like what Farooq [3] did. However, due to its computational time, it is only applicable
for graphs with few nodes. Therefore, for that purpose, Dijkstra’s algorithm was employed
(see, Appendix C). For an easy access to the shortest distances found by the algorithm, a
shortest distance matrix is constructed such that an entry \(d_{ij}\) in the matrix would indicate
the shortest distance between the required node \(i\) and the required node \(j\). As an example, the
shortest distance matrix between the required nodes in Figure 6.11 would be the same as in
Table 6.1. The shortest distances between the required nodes are used in deciding the length
of every candidate nominated as the shortest cycle.

Table 6.1 Example of a shortest distance matrix

<table>
<thead>
<tr>
<th></th>
<th>(n_1)</th>
<th>(n_3)</th>
<th>(n_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_1)</td>
<td>0</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>(n_3)</td>
<td>10</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>(n_6)</td>
<td>13</td>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>
It is also required to keep the track of the shortest paths in a list or an array. Those tracks would be used in composing the detailed path of a near the shortest cycle after it is decided by the simulated annealing algorithm. A 2-D array of tuples is built up for saving the tracks, similar to the shortest distance matrix. In that array, entry $p_{ij}$ would be the shortest path between the required node $i$ and the required node $j$. As an example, Table 6.2 would be the shortest path array of the graph in Figure 6.11.

<table>
<thead>
<tr>
<th></th>
<th>$n_1$</th>
<th>$n_3$</th>
<th>$n_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>(</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>$n_3$</td>
<td>(</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>$n_6$</td>
<td>(</td>
<td>(</td>
<td>(</td>
</tr>
</tbody>
</table>

As soon as the transformation is done, the graph would be fed into a simulated annealing algorithm. The principles of the algorithm are similar to what has been described in Appendix B. The initial candidate, mentioned in the description, would be generated by a random arrangement of required nodes. Also, for generating every new candidate (i.e., a neighbor) from a current candidate, one of the two following approaches for permutation would be taken randomly (by a probability, such that $p = 0.5$):

1. Swapping two nodes in the current solution tuple, randomly
2. Reversing order of the nodes between two random points of the solution tuple

Moreover, candidates would be evaluated by simply checking their length. The length is sum of the shortest distances between the adjacent required nodes in the solution tuple (including the distance between the first and the last required nodes).

As soon as a near the shortest cycle is found and it is transformed into the detailed form (i.e., the optional nodes are included), its length and risk (which are determined by using Eq. 4.3, and 4.5) would be used in calculating the lower and upper bounds (using Eq. 4.6, 4.7 and 4.8).

### 6.3.5 Cycle Validation

This part explains the items which are checked in validating a candidate in the main search (i.e., the DFS search). Due to the usage of the conditional directions, there are only a few items to be checked for the validation. The three characteristics for which a partial or a complete cycle is rejected are listed below:

1. Transition through an edge more than twice (see, Figure 6.12a)
2. Transition through an edge more than once, in the same direction (see, Figure 6.12b)
3. A path forming a loop, not including any required node (see, Figure 6.12c)

![Diagram of invalid cycles](image)

Figure 6.12 Examples of invalid cycles with (a) characteristic 1, (b) characteristic 2 and (c) characteristic 3

It is not required to show a proof for the first characteristic, since based on the definition of a valid cycle, inclusion of an edge more than twice is not allowed.

For cycles with the second characteristic, it will be shown that such a cycle could not be a most reasonable. To this end, assume \( N = \{n_1, n_2, ..., n_m\} \) is the set of all nodes of graph \( G \) and \( S = (s_{n_1}, s_{n_2}, ..., s_{n_i}) \) is a partial solution including two transitions through edge \( e \), where every node \( s_n \) in \( S \) corresponds to a node in \( N \). Since there are two transitions through \( e \) in the same direction, there must be nodes \( s_{n_k}, s_{n_{k+1}}, s_{n_{k+c}} \) and \( s_{n_{k+c+1}} \) in \( S \), such that:

\[
e = (s_{n_k}, s_{n_{k+1}}) = (s_{n_{k+c}}, s_{n_{k+c+1}}), \text{ where } 1 < k, 1 < c, \text{ and } k + c + 1 < i
\]

So, \( S \) could be re-defined as:

\[
S = (s_{n_1}, s_{n_2}, ..., s_{n_k}, s_{n_{k+1}}, ..., s_{n_{k+c}}, s_{n_{k+c+1}}, ..., s_{n_i})
\]

, where \( s_{n_k} = s_{n_{k+c}} \) and \( s_{n_{k+1}} = s_{n_{k+c+1}} \)
Consider path $P$ in $S$ from node $sn_k$ to node $sn_{k+c}$, such that $P = (sn_k, sn_{k+1}, ..., sn_{k+c})$. Since $sn_k = sn_{k+c}$, $P$ could be replaced with its reversion $P' = (sn_{k+c}, ..., sn_{k+1}, sn_k)$ and produce solution $S'$ as below:

$$S' = (sn_1, sn_2, ..., sn_{k+c}, sn_{k+1}, sn_k, sn_{k+c+1}, sn_{k+c+2}, ..., sn_l)$$

However, as $sn_{k+1} = sn_{k+c+1}$, there would be an unnecessary return in $S'$ at $sn_k$ (through $e$) without any inclusion of a new node, because $sn_k$ would have been already included in $S'$ ($sn_k = sn_{k+c}$). Therefore, if $e$ is removed from $S'$, the new solution $S''$ would have a shorter length and risk than $S'$; because the path of $S''$ would be a sub-path of $S$. So, $S''$ would be a better solution than $S$ and consequently $S$ could not be most reasonable.

Although, it might intuitively be accepted that cycles with the third characteristic are improper, it will be proved below for certainty. To this end, the definition of a “loop in a path” should be given first, as follows. A loop in a path is two-time occurrence of a node in the path with some nodes (other than that node) in between. Now, assume solution $S$ containing a loop incident at node $n$ with no required nodes between the two occurrences of $n$. By removing all the nodes between the two occurrences of $n$ and merging the two occurrences into one, one could produce solution $S'$ which is shorter than $S$ (because $S' \subset S$) with the same risk. Hence, $S$ could not be most reasonable.

6.3.6 Heuristic Cost Estimation

The cost (i.e., length and risk) of a complete cycle is always higher than a partial cycle from which that complete cycle is derived. Thus, if that cost could be estimated by looking at partial cycles beforehand, further investigations of partial cycles which end in expensive cycles could be avoided. Since inclusion of an edge in a cycle does not independently affect the risk of a cycle, such estimation for the risk is not possible. However, fortunately, it is possible to estimate a minimum length for cycles which are derived from a partial cycle, by a heuristic approach in advance.

Basically, a partial cycle is a path which connects some or all of required nodes and usually its head and tail points (i.e., two end points) are not the same. To have such a path complete as a valid cycle, any remaining required node should be included and/or the two ends must be linked together through a complementary path. A lower bound for the length of cycles which are derived from a partial cycle could be determined by adding up the following two lengths:

1. The length of the partial cycle
2. The length of a shortest complementary path which links both any remaining required nodes and the end points of the partial cycle
Since finding that shortest complementary path is itself NP-hard, the following approach could be an alternative. For that, first, the two Manhattan distances\(^\text{19}\) between the end points of a partial cycle and every remaining required node (see, Figure 6.13) will be obtained. Then, those two distances would be summed up together, for every remaining required node separately. A sum with the greatest value among others would be an estimate for the length of the shortest complementary path. Adding that estimate to the length of the partial cycle would result the required lower bound. For those partial cycles which are already linking all required nodes, the estimate for the complementary path would be the Manhattan distance between the two end points of the partial cycle.

\[ \text{Distance from the head of a partial cycle to a remaining required node} \]
\[ \text{Distance from the tail of a partial cycle to a remaining required node} \]

**Figure 6.13** Measuring Manhattan distances for heuristic estimation of the final length of a partial solution

As stated earlier, estimating a risk in advance is not possible. Therefore, in evaluating every partial cycle, the length would be an estimate for a complete cycle, but the risk would be the exact current risk of the partial cycle.

This chapter finishes here as all the techniques used in this work for a sophisticated guided search were introduced. In the next chapter, the result of the experiment with the two sample sets will be presented and discussed.

\(^{19}\) 3-D Manhattan distances using the coordinates
7 Results and Discussions

In Chapter 3, the sample set, provided by the company for this work, was introduced. By using the five sets of objects and setting different values for the acceptable extra length parameter, 11 different problem instances (i.e., scenarios) were set up as listed in Table 7.1 (see, column 1-4). For each set of objects, two (for one set three) different values for the parameter were set to see its effect on the quality of the solutions and the relative computation time. In all the tables in this chapter, those problem instances which were derived from a same set of objects are grouped by using dashed lines (note that the difference between the instances in a same group would be only different values for the parameter).

For each problem instance, three different types of cycles are provided for comparison purpose. All of those types were already described at several parts of this work (specifically in Section 2.2), but below they will shortly be reviewed as a reminder.

One type is near the shortest cycle for which the only concern is the length of the cable. This type of cycle is used not only for a comparison purpose, but in deciding lower and upper bounds in a search for other two types of cycle. A most reliable cycle is another type which is provided and for which the only concern is the risk. The risk of a cycle of that type is least among other candidates. A most reasonable cycle is the last type which is a compromise between the length and risk. A cycle of that type has minimum overall ratio (defined in Chapter 4) among other candidates.

To present the result, first, the properties (i.e., length and risk) of the three types of cycle for the 11 problem instances are listed in Table 7.1 (column 5-10). Moreover, for a better observation of possible changes between the properties of a near the shortest cycle and the other two types of cycle, a separate table (i.e., Table 7.2) indicating those changes (in percentage) is provided. Also, Table 7.2 is illustrated by two charts: one for showing changes in length and risk with most reasonable cycles (see, Figure 7.1) and one for showing changes in length and risk with most reliable cycles (see, Figure 7.2), both in comparison with corresponding near the shortest cycles.

By looking at Table 7.1, one could realize that two or all of the three types of cycle are identical for some problem instances (due to either the arrangement of required nodes, or the value which was set for the acceptable extra length parameter). For instance, all the three types of cycle are the same for problem instance 6, due to the value of acceptable extra length parameter which does not allow a cycle to be more than 30m longer than the near the shortest cycle. As soon as the parameter is increased to 50m for the same set of required nodes (resulting problem instance 7), the most reasonable and reliable cycles would differ from the near the shortest one. Since an increment of the parameter from 50 to 70m does not provide any more changes in the solutions for the 1st, 3rd, 4th and 5th groups of problem instances (due
to the arrangement of their required nodes), the result of such an increment for the 2nd group is only presented. As could be observed, by such an increment, the most reliable cycle differs from the most reasonable cycle for problem instance 5.

Although by increasing the value of the acceptable extra length parameter one allow a cycle to be longer in return for a lower risk (which is effective to some extent for some instances), the run-times undesirably increases as well (this would be showed and discussed later). Moreover, in reality, the company is not interested in solutions which are very longer than the shortest one, even if the solution has no risk. Thus, it is suggested to start a search with a small value for the parameter and then try with larger values if the solution is not satisfactory.

As stated before, Table 7.2, Figure 7.1 and Figure 7.2 depict possible changes in the properties of most reasonable and reliable cycles compared to near the shortest cycles. The trade-off between length and risk is obvious in most of the cases.

In Figure 7.1, it could be observed that other than for problem instance 6 and 7, the most reasonable solutions considerably reduce the risk in return for a little longer length compared to the their corresponding near the shortest solutions. The reason of seeing different patterns for problem instance 6 and 7 is that the arrangement of the required nodes makes it impossible to form a cycle which is more reliable but not very longer than the shortest one.

**Table 7.1 Eleven problem instances and properties of three different types of solution to them**

<table>
<thead>
<tr>
<th>Instance No.</th>
<th>No. of Edges</th>
<th>No. of Nodes*</th>
<th>AEL** (m)</th>
<th>Near the Shortest Cycle</th>
<th>Most reasonable Cycle</th>
<th>Most reliable Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Length (m)</td>
<td>Risk (mO***)</td>
<td>Length (m)</td>
</tr>
<tr>
<td>1</td>
<td>109</td>
<td>5/93</td>
<td>30</td>
<td>209.92</td>
<td>53.68</td>
<td>235.66</td>
</tr>
<tr>
<td>2</td>
<td>109</td>
<td>5/93</td>
<td>50</td>
<td>209.92</td>
<td>53.68</td>
<td>235.66</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>7/95</td>
<td>30</td>
<td>43.56</td>
<td>103.23</td>
<td>43.56</td>
</tr>
<tr>
<td>4</td>
<td>111</td>
<td>7/95</td>
<td>50</td>
<td>43.56</td>
<td>103.23</td>
<td>43.56</td>
</tr>
<tr>
<td>5</td>
<td>111</td>
<td>7/95</td>
<td>70</td>
<td>43.56</td>
<td>103.23</td>
<td>43.56</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>8/94</td>
<td>30</td>
<td>67.50</td>
<td>99.92</td>
<td>67.50</td>
</tr>
<tr>
<td>7</td>
<td>110</td>
<td>8/94</td>
<td>50</td>
<td>67.50</td>
<td>99.92</td>
<td>108.90</td>
</tr>
<tr>
<td>8</td>
<td>114</td>
<td>10/98</td>
<td>30</td>
<td>217.20</td>
<td>296.63</td>
<td>219.40</td>
</tr>
<tr>
<td>9</td>
<td>114</td>
<td>10/98</td>
<td>50</td>
<td>217.20</td>
<td>296.63</td>
<td>219.40</td>
</tr>
<tr>
<td>10</td>
<td>117</td>
<td>14/101</td>
<td>30</td>
<td>170.69</td>
<td>166.85</td>
<td>188.80</td>
</tr>
<tr>
<td>11</td>
<td>117</td>
<td>14/101</td>
<td>50</td>
<td>170.69</td>
<td>166.85</td>
<td>188.80</td>
</tr>
</tbody>
</table>

*Number of required / total nodes  
**Acceptable Extra Length  
***Remind mO is the unit of risk, where m represents meter and O object (see, Chapter 4)
However, even for that instance, the amount of reduced risk is about 39% more than the amount of increased length.

Table 7.2 Changes in length and risk with most reasonable and most reliable cycles in comparison with corresponding near the shortest cycles

<table>
<thead>
<tr>
<th>Instance No.</th>
<th>Most Reasonable Cycle</th>
<th>Most Reliable Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Change in Length (%)</td>
<td>Change in Risk (%)</td>
</tr>
<tr>
<td>1</td>
<td>12.26</td>
<td>-54.96</td>
</tr>
<tr>
<td>2</td>
<td>12.26</td>
<td>-54.96</td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>-13.56</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>-13.56</td>
</tr>
<tr>
<td>5</td>
<td>0.00</td>
<td>-13.56</td>
</tr>
<tr>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>61.33</td>
<td>-100.00</td>
</tr>
<tr>
<td>8</td>
<td>1.01</td>
<td>-77.90</td>
</tr>
<tr>
<td>9</td>
<td>1.01</td>
<td>-77.90</td>
</tr>
<tr>
<td>10</td>
<td>10.61</td>
<td>-76.05</td>
</tr>
<tr>
<td>11</td>
<td>10.61</td>
<td>-76.05</td>
</tr>
</tbody>
</table>

Figure 7.1 Illustration of changes in length and risk with most reasonable cycles in comparison with corresponding near the shortest cycles (illustration of column 2-3 of Table 7.2)
In Figure 7.2, a large deterioration in the length of the solution to problem instance 5 is seen. Moreover, the risk of the solutions of this type to problem instance 9 and 11 is even less than the most reasonable type (in return for a longer length). The changes sound reasonable due to the major concern for a most reliable solution which is reliability.

Figure 7.2 Illustration of changes in length and risk with most reliable cycles in comparison with corresponding near the shortest cycles (illustration of column 4-5 of Table 7.2)

Now, let us go through the run-times of the experiments for obtaining the solutions. The tests performed on a computer with Intel® Core™ 2 Duo 2.10 GHz processor, 3.00 GB of RAM and Microsoft® Windows Vista™ OS and the implementation was done in Matlab, version 7.12.0.635 (R2011a).

In general, the run-times are divided into two categories as follows:

1. The run-time of the simulated annealing process for obtaining a near the shortest cycle
2. The overall run-time of obtaining both the most reasonable and most reliable cycles

In Table 7.3, the measured run-times of both categories for every problem instance (when all the proposed techniques are used) is presented. Although, the result shows the run-times are highly related to the number of nodes and edges of the instances as well as the value of the acceptable extra length parameter, a different pattern could be observed for some of the instances. That is, the run-time for some of the instances is more than some other instances which have higher number of nodes or edges. For example, the run-time for the problem

---

20 Since the two types of the solution are found during the same search, for both types one run-time is measured and considered.
instance 1 (with 5 required nodes and 109 edges) is 4.2 seconds, while it is 0.44 seconds for problem instance 6 (with 8 required nodes and 110 edges). The reason is that a run-time depends on not only the topology of a graph, but also the geometry of the components of a graph. This means, for example, the run-time for an instance in which all required nodes are close to each other could be shorter than an instance in which some or all of required nodes are far from each other (with the same number of required nodes and edges). This could easily be observed for problem instance 1 and 6 by looking at the length of their near the shortest cycles in Table 7.1.

Table 7.3 Run-times of obtaining three different types of solutions to 11 problem instances by using all proposed techniques

<table>
<thead>
<tr>
<th>Instance No.</th>
<th>Near the Shortest Cycle</th>
<th>Most Reasonable and Most Reliable Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time Taken (sec.)</td>
<td>Time Taken (sec.)</td>
</tr>
<tr>
<td>1</td>
<td>3.386249</td>
<td>4.288419</td>
</tr>
<tr>
<td>2</td>
<td>3.444732</td>
<td>9.472564</td>
</tr>
<tr>
<td>3</td>
<td>3.803044</td>
<td>0.337423</td>
</tr>
<tr>
<td>4</td>
<td>2.946389</td>
<td>0.720021</td>
</tr>
<tr>
<td>5</td>
<td>3.111949</td>
<td>1.471676</td>
</tr>
<tr>
<td>6</td>
<td>3.113646</td>
<td>0.440206</td>
</tr>
<tr>
<td>7</td>
<td>3.038961</td>
<td>0.493604</td>
</tr>
<tr>
<td>8</td>
<td>3.060232</td>
<td>23.485951</td>
</tr>
<tr>
<td>9</td>
<td>3.482628</td>
<td>43.272453</td>
</tr>
<tr>
<td>10</td>
<td>2.986975</td>
<td>95.750013</td>
</tr>
<tr>
<td>11</td>
<td>3.177213</td>
<td>240.849341</td>
</tr>
</tbody>
</table>

To show that how much using the proposed techniques speed up a brute-force search, new tests were performed without using one of the following four techniques (which are the main ones among the discussed techniques) in each experiment:

- *Technique 1*: the conditional directions
- *Technique 2*: the inhibition of a transition through an edge more than once in the same direction
- *Technique 3*: the inhibition of a loop which does not include any required node
- *Technique 4*: the heuristic length estimation
Totally, 44 new experiments were done for the 11 problem instances (i.e., four tests per instance). Figure 7.3 illustrates the result of the tests (i.e., run-times) along with the previous run-times\textsuperscript{21} (when all the techniques are used) for a comparison purpose.

![Figure 7.3 Illustration of time taken for obtaining solutions (most reasonable and reliable) by exploiting all and not exploiting each of the proposed techniques](image)

As the figure depicts, the search becomes slower for most of the instances without exploiting each of the four techniques. However, it becomes a bit faster for some of the smaller instances when technique 2 is not used (due to the computational burden of the technique). The growth in run-times is more obvious for the instances with a greater number of nodes. For problem instance 1 and 2 the most effective techniques are 3 and 4, while for other instances the most effective ones are almost technique 3 and 4.

It is important to know that exploiting none of the main techniques causes a vast increment in the run-time of the searches which is not a equal to the sum of the increments when every technique separately is not used. Performing tests to show it for all of the problem instances would be cumbersome and very time-consuming. As an example, such a test only for a small instance of the problem (with 41 edges, 26 optional nodes, 10 required nodes and setting 30m for the acceptable extra length parameter) took 2874.71 seconds, compared to 0.9 seconds when all the techniques were used.

Now, as the effectiveness of the techniques in reducing the run-time of the searches was shown, one could think of applying some or all of these techniques in solving other problems with similar properties. For example, the conditional directions as well as the other

\textsuperscript{21} Since the time taken for finding a near the shortest cycle is unchanged in every test, it is excluded from the result.
techniques introduced in the work seem to be also useful in solving larger instances of the order-picking problem in a warehouse to optimum. Also, the method of using an approximate algorithm in setting bounds for an exact algorithm might be a good idea in solving larger instances of problems like the TSP, to optimum. Moreover, the introduced heuristic length estimation might be tested for variety of routing problems.
8 Conclusion

This work dealt with a routing problem of a cable cycle for connecting a set of objects through a given set of routes (in cooperation with AB Tändkulan). For solving the problem, both financial and technical concerns were required to be taken into account. Based on the two concerns, length and risk factors for evaluating a cycle were defined and formulized. Also, the raw data transformation into a form of a graph to be more convenient for solving the problem.

Since the problem is NP-hard, the first attempt for solving that, was on developing an approximate algorithm. However, due to the nature of the problem the attempt was unsuccessful. Also, according to author’s study, the problem could not be seen as a variation of the TSP, due to the risk factor.

The second attempt was on finding cycles for subsets of objects and merging them together as a solution. That idea also failed due to technical issues with solutions which were not reasonable in some cases. However, the method works well when the only concern is the risk.

The final successful attempt was on developing an exact algorithm which exploits an approximate algorithm (a simulated annealing) to set bounds for the solutions. A depth-first search strategy was taken with a sophisticated backtracking approach. Moreover, a new kind of a graph was developed due to the fact that many invalid and nonreasonable solutions could be detected in the middle of a search. The graph could be considered as one of the prominent parts of the work. To prepare such a graph and do a guided search, signs and conditional directions are added to the nodes of an original undirected graph of the problem. Using those directions and signs helps avoid further expansions of undesired partial solutions and have lesser computations in the validation process.

Also, a heuristic approach was proposed to estimate the length of feasible solutions which are derived from a partial solution, in advance. It was shown, the search tree could be pruned, considerably, by means of that heuristic. The approach is very useful for an early stopping of a further consideration of the branches of a tree ending to expensive leaves.

The method was tested for 11 problem instances. The result showed the method could provide satisfactory solutions to the problem within reasonable amount of time. It also showed the proposed techniques for reducing the search-space are very effective and could make a brute-force search possible for problems with similar scale. Although the method was tested only for the problem in this work, it sounds to work also for solving larger instances of similar NP-hard problems to which an optimal solution is unknown yet. However, of course, there is still no method for solving very large instances of NP-hard problems to optimum in reasonable amount of time.
Finally, in this work, a comparison with other works could not be carried out because the problem had not been addressed and solved by others before. However, this work could be used as a basis for future works.
Appendix A: The Travelling Salesman Problem

The Travelling Salesman Problem is probably one of the most famous combinatorial optimization problems [21]. It arose in the eighteenth century and it was studied seriously in the late 1940s and early 1950s as an NP-hard problem. Given $N$ (finite) cities with the cost of travel from one to another (e.g., the distance between each pair), a travelling salesman needs to find a way for visiting each city exactly once and returning to the starting city, with minimum total cost. [22] The order of the cities is called a tour. If the problem is represented with a graph where the nodes correspond to the cities and the edges (weighted) correspond to the roads between the cities, the goal is to find the shortest Hamiltonian cycle.

An exact solution to this problem is an enumeration of all possible permutations of the cities and selection of the tour with the minimum cost as the best solution. However, the computational cost of finding the exact solution is too much (i.e., its time complexity is $O(n!)$) for a large number of cities. There are many heuristic and approximation techniques to find a solution which is near the optimal one and acceptable within a reasonable amount of time. Some examples are simulated annealing (SA), genetic algorithms (GA), swarm intelligence algorithms, and so on.

Solutions to the TSP are applicable to other optimization problems which are similar to the TSP and can be seen as its variation. One example of these variations can be the minimization of the length of a cable ring which connects specific nodes in a building or links set of sites in a SONET network [23].
Appendix B: Simulated Annealing

Simulated annealing [24][25] is a stochastic search that unlike a local search does not get stuck in local optima. The terms and the method is an analogy with the annealing in thermodynamic and metallurgy, which is a process of heating and cooling of materials for having crystal configuration with minimum energy. The process starts by heating up a solid to a maximum high temperature to have a liquid in which particles move freely. Then, the liquid slowly gets cooled to shape a perfect crystal. By this, atoms arrange themselves in lower energy ground state.[26][27][28]

Simulation of the annealing is widely used in optimization of complex problems to which an exact solution is computationally very time-consuming. It has also been a favourable method for the TSP family.

There are many different versions of the method, but in principle they are more or less similar. Generally, the procedure gets started at high temperature (i.e., a parameter for controlling the acceptance of a new candidate) with an initial candidate. On every iteration, a neighbourhood function generates a new neighbour of a current candidate. There is also an objective or cost function which evaluates every new neighbour. In case of minimization, if the cost of a new neighbour is lower than the current candidate, the new neighbour is replaced with the current candidate as a better solution; otherwise the new neighbour is accepted randomly. The probability of the random acceptance is calculated by Eq. AC.1:

\[ p = e^{-\frac{eval(\text{current candidate}) - eval(\text{neighbor})}{T}} \]  

(AC.1)

where \( eval \) is the evaluation function and \( T \) the current temperature. As the procedure progresses the temperature is decreased until a stopping condition is reached.
Dijkstra’s Algorithm [29] is a well-known algorithm for the shortest path problem. The shortest path problem can be defined as follows: given a graph with sets of nodes and weighted edges (directed, undirected or mixed). It is desired to find the minimum cost path from a source node to a destination node, where cost is the sum of the weights of the edges included in the path. Solutions to the problem have many applications in web mapping, networking, transportation, and so on. Dijkstra’s algorithm is able to find all the shortest paths from a source node to all other nodes (reachable from the source) in a graph with non-negative weights. The final solution is in the form of a tree and any sub-path of the tree is itself a shortest path [30]. The algorithm is fast and efficient and rather easy to be implemented. The strategy of the search is greedy [31] and the final solution is guaranteed to be optimal. The complexity of the simplest implementation of the algorithm is $O(n^2)$ where $n$ is the number of nodes in a graph.

In general, the algorithm divides nodes of an input graph into two groups of marked and unmarked. The list of the marked nodes in the initial state includes only the source node and the list of the unmarked nodes includes the rest. It is also needed to construct a list for keeping the shortest distances found so far from the source to every other node. The list is initialized by zero for the distance from the source to itself, and infinite for the distances to other nodes. At every step, an unmarked node with the smallest value in the shortest distance list is marked and removed from the unmarked node list. Then, the distance of the path from the source to every adjacent node to the marked node (via the marked node) is compared with the corresponding distance in the shortest distance list. If any of those new paths from the source to the adjacent nodes is shorter, the new distance is replaced with an old one. A list for keeping track of the shortest paths can also be constructed. This list indicates the preceding node of every node in the shortest paths. Therefore to know the shortest path from source to destination node, one should move backward in the list, starting from the destination node, and check every previous node to reach to the source.
Appendix D: Procedure of Guided Search

procedure adaptedbacktrackingBB(graph, lengthLowerBound, ... lengthUpperBound, riskUpperBound)
create a stack S
mostReasonableRatio ← \( \infty \)
minReliableCycleRisk ← riskUpperBound
minReliableCycleLength ← \( \infty \)
notExpandedPath ← primary node from graph
S.push(notExpandedPath)
while S is not empty do
    notExpandedPath ← S.pop
    if notExpandedPath is cycle and no required node is remaining
        candidateLength ← length(notExpandedPath)
candidateRisk ← risk(notExpandedPath)
candidateOverallRatio ← overallRatio(candidateLength, ... candidateRisk, lengthLowerBound, riskUpperBound)
if candidateOverallRatio < mostReasonableRatio
    mostReasonableCycle ← notExpandedPath
    mostReasonableRatio ← candidateOverallRatio
if candidateRisk < minReliableCycleRisk or ... (candidateRisk == minReliableCycleRisk and ... candidateLength < minReliableCycleLength)
    mostReliableCycle ← notExpandedPath
    minReliableCycleRisk ← candidateRisk
    minReliableCycleLength ← candidateLength
else
    currentNode ← the last node in notExpandedPath
    previousNode ← one node before the last node in ... notExpandedPath
    nextNodes ← next adjacent nodes to currentNode ... considering directions and previousNode
    for every node n in nextNodes do
        expandedPath ← add n to notExpandedPath
        if isvalid(expandedPath)
            estimatedLength ← ...
            estimateLength(expandedPath)
            evaluatedRisk ← risk(expandedPath)
            expandedPathOverallRatio ← ...
            overallRatio(estimatedLength, ... evaluatedRisk, lengthLowerBound, ... riskUpperBound)
            if estimatedLength ≤ lengthUpperBound and ...
                evaluatedRisk ≤ riskUpperBound ... and (expandedPathOverallRatio < ... mostReasonableOverallRatio or ... evaluatedRisk ≤ ... minReliableOverallRisk)
S.push(expandedPath)
return(mostReasonableCycle, mostReliableCycle)
References


Architecture,” Systems, Man and Cybernetics, Part A: Systems and Humans,


by Meta-Heuristics,” in Computing Tools for Modeling, Optimization and


