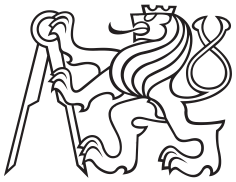
Bachelor Project



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Meta-heuristics for routing problems

Jan Mikula

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
May 2018

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Mikula Jan** Personal ID number: **457197**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Systems and Control**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Meta-heuristics for routing problems

Bachelor's thesis title in Czech:

Meta-heuristiky pro směrovací problémy

Guidelines:

1. Get acquainted with the Traveling Deliveryman Problem and the Graph Search Problem.
2. Get acquainted with meta-heuristics for routing problems (tabu search, Variable Neighborhood Search, Greedy randomized adaptive search procedure, etc.).
3. Design a meta-heuristic for the Traveling Deliveryman Problem and the Graph Search Problem and implement it.
4. Evaluate experimentally properties of the implemented algorithm. Describe and discuss obtained results.

Bibliography / sources:

[1] Kulich, M.- Miranda-Bront, J. - Přeučil, L.: A meta-heuristic based goal-selection strategy for mobile robot search in an unknown environment. Computers & Operations Research. vol 84, August 2017, pp. 178-187.
[2] N. Mladenović, D. Urošević, and S. Hanafi, Variable neighborhood search for the travelling deliveryman problem, 4OR, pp. 1-17, 2012.
[3] M. M. Silva, A. Subramanian, T. Vidal, and L. S. Ochi, A simple and effective metaheuristic for the Minimum Latency Problem, European Journal of Operational Research, vol. 221, pp. 513-520, Sept. 2012.

Name and workplace of bachelor's thesis supervisor:

RNDr. Miroslav Kulich, Ph.D., Intelligent and Mobile Robotics, CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **16.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to express my gratitude to my supervisor RNDr. Miroslav Kulich, Ph.D. for his invaluable guidance, patience, and advice. I also wish to thank my family for their endless support during my studies.

Declaration

I hereby declare that I have completed this thesis independently and that all the used sources are included in the list of references, in accordance with the Methodological Instructions on Ethical Principles in the Preparation of University Theses.

In Prague, May 25th, 2018

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. 5. 2018

.....

Abstract

The Graph Search Problem (GSP) together with the Traveling Deliveryman Problem (TDP) can be seen as an extension of the famous Traveling Salesman Problem (TSP), i.e., a problem of finding the shortest possible route that visits every city of a given set and returns to the one where it started. These, sometimes called the routing problems, belong to a category of difficult combinatorial optimization problems. In this thesis, we address the GSP with the intention to deploy it in robotic planning, which entails high demands on feasible computing time. We develop a meta-heuristics for the GSP based on Variable Neighborhood Search (VNS) and evaluate its properties experimentally. Our proposed method meets the requirements of being significantly faster than the reference, and at the same time, it delivers solutions with higher average quality. We demonstrate the usage of our method in the mobile robot search, which turns out to be a stand-alone problem to solve. The second half of this thesis is devoted to the Single Robot Search for a Stationary Object in a Known Environment (SRSSK). At the end, we execute several simulations in a robotic simulator.

Keywords:
Routing problems
Graph Search Problem
Variable Neighborhood Search
Mobile robotics
Robotic search
ROS

Supervisor:
RNDr. Miroslav Kulich, Ph.D.
Intelligent and Mobile Robotics,
CIIRC
Jugoslávských partyzánů 1580/3
160 00 Praha 6

Abstrakt

Problém hledání v grafu (anglicky Graph Search Problem - GSP) společně s problémem obchodního doručovatele lze považovat za rozšíření dobře známého problému obchodního cestovatele, který hledá nejkratší možnou cestu, jež mu umožní navštívit všechna města z dané množiny a skončit zase tam, kde začal. Tyto, někdy nazývané směrovací problémy, patří do kategorie obtížných kombinatorických optimalizačních problémů. V této práci se budeme zabývat GSP, s úmyslem ho následně využít v robotickém hledání, které nastavuje velké požadavky na schůdný výpočetní čas. Vyvineme metaheuristiku pro GSP založenou na metodě hledání v proměnlivém sousedství (anglicky Variable Neighborhood Search - VNS) a experimentálně vyhodnotíme její vlastnosti. Navrhovaná metoda bude výrazně rychlejší než referenční metody a zároveň bude dávat řešení s vyšší průměrnou kvalitou. Použití navrhované metody budeme demonstrovat v problému robotického plánování. Druhá polovina této práce bude tedy věnovaná hledání robotu ve známém prostředí s cílem nalézt nehybný objekt zájmu (anglicky Single Robot Search for a Stationary Object in a Known Environment - SRSSK). Nakonec provedeme několik simulací v robotickém simulátoru.

Klíčová slova:
Směrovací problémy
Graph Search Problem
Variable Neighborhood Search
Mobilní robotika
Robotické hledání
ROS

Překlad názvu:
Meta-heuristiky pro směrovací problémy

Contents

Figures

Tables

Chapter 1

Preliminaries

1.1 Introduction, goals, structure

The Graph Search Problem (GSP) together with the Traveling Deliveryman Problem (TDP) can be seen as an extension of the famous Traveling Salesman Problem (TSP) - a problem of finding the shortest possible route that visits every city of a given set and returns to the one where it started. These are NP-hard belonging to a category of difficult combinatorial optimization problems challenging researchers of different fields for more than 50 past years. Finding a solution to those problems by force (i.e., constructing every possible route and picking the shortest one amongst them) is theoretically possible but somewhat impractical. When the number of given cities is large (i.e., several dozens) the computational time needed to construct the whole space of possible solutions becomes unbearable. The key is, therefore, developing a time-efficient algorithm for searching the possible solution space for near-optimal solutions. The requirements on solution quality and computational time often depend on an application. The TSP and similar problems find their use in many different fields. An interested reader can find some of them in Cook's *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation* [7] as well as in some articles cited in this thesis. Let us mention: transport and delivery of goods, computer wiring, vehicle routing or planning in almost any domain.

We address the GSP with the intention to deploy it in robotic planning which is usually a real-time application, so the methods used there need to be really fast. Our approach to the problem will be mostly based on recent paper Kulich, Bront & Přeučil [16] titled: *A meta-heuristic based goal-selection strategy for a mobile robot search in an unknown environment*, where they introduce several GSP solving methods as well as an entire framework suited for the robotic application.

The primary objective of ours is designing a meta-heuristics for the GSP, implementing it and evaluating its properties experimentally. We will use two

of the proposed methods in [16] as a reference. We put the following demand on our proposed method: to be significantly faster than the reference while keeping an average solution quality on a similar level.

Our secondary objective is to demonstrate the usage of our method in a robotic application such as the mobile robot search. We do not demand the usage to be strictly real-time but rather to aim in that direction and offer possibilities for extensions in the future. The framework will be inspired by [16] but simplified, and we call it the Single Robot Search for a Stationary Object in a Known Environment (SRSSK). Although the act of planning will be performed off-line (i.e., with the robot standing in place, not during the process of searching), we will try to highlight the benefits of faster methods nevertheless. We will have a taste of different approaches on the way and compare and discuss them at the end. We will propose possible extensions such as the on-line planning and improvement of the used procedures. Finally, we will execute a set of simulations with the help of Robot Operation System (ROS) and other software tools.

This thesis is divided into two main parts: Part I devoted to the GSP and Part II addressing the SRSSK. Both have their own problem definitions (in Chapters 2 and 5 respectively), solution approach descriptions (in Chapters 3 and 6 respectively) and computational results appearance and discussion (in Chapters 4 and 7 respectively). Chapter 1 is introductory and it is common for both parts as well as conclusions in Chapter 8. In Chapter 9 we summarize some ideas of improvement and potential future research.

1.2 Subject background

Traveling Salesman Problem. The TSP is a well-known combinatorial optimization problem with its origins in practical business trips planning. Its complete historical development can be found in Cook [7] as well as why it is so crucial in operational research. As stated in Cook, TSP commanded much attention from mathematicians and theoretical computer scientists in past 50 or more years. It has a variety of modifications and extensions that in contrast with the original problem only small subset of researchers was dealing with. The TDP and its weighted variant the GSP are those amongst them. TSP can be in the mathematical field of graph theory modeled as an undirected weighted graph, where cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. A Hamiltonian path is such a path that visits each vertex exactly ones. A Hamiltonian cycle is a Hamiltonian path that is a cycle[1]. We paraphrase the definition from Cook.

Definition 1.1. The TSP can be defined as follows. Given a complete undirected weighted graph find a Hamiltonian cycle with the least weight.

[1] Source: `http://mathworld.wolfram.com/HamiltonianPath.html`.

Traveling Deliveryman Problem. The TDP is a modification of the TSP with the only difference in motivation of the traveler. While the salesman only wants to save money on his journey (by taking the shortest path), the deliveryman is there primarily for the benefit of customers. His ambition is to serve them all while minimizing the total time of them waiting to be served. Also, the last part of his journey when he needs to travel back to his starting point is often not accounted (regarding graph theory we are talking about finding the shortest Hamiltonian path, not a cycle). Let us notice that unlike GSP TDP is a problem, that most of the operational research community is familiar with. It can be found under various names in the literature, e.g., Traveling Repairman Problem in [33], Cumulative Traveling Salesman Problem in [4], School Bus Driver Problem in [5] or Minimum Latency Problem in [32].

Graph Search Problem. The GSP can be seen as a Weighted TDP. Every customer has assigned weight which corresponds to his lucrativeness from the deliveryman's point of view. In other words, the deliveryman wants to serve the customers with higher weight first and after them those with lower. He, therefore, aims to minimize the sum of the time each customer waits times his weight. GSP was introduced in Koutsoupias et al. [15], and its first applications were built around the area of web searching. In every one of the mentioned problems appear some travel agents. This connection between TSP, DSP and GSP (in case of the GSP they call the agents spiders) is discussed in Ausiello et al. [3], together with some approximation schemes for the problems. No more progress on the GSP can be found in the related literature with the only exception of Kulich, Bront & Přeučil [16]. We paraphrase their definitions.

Definition 1.2. The GSP is described by a complete undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges between the vertices. G has a distance function $d(i, j)$, $d : V \times V \to \mathbb{R}$ for every $(i, j) \in E$, $i \neq j$ and a probability p_i that some required information is at vertex $i \in V$. The objective of the GSP is to find a Hamiltonian path that minimizes the expected time to find the required information.

Definition 1.3. The TDP is the GSP for which applies $p_i = p_j$ for every $i, j \in V$.

Heuristics. Heuristics is any technique designed to solve a problem more quickly than existing exact methods or to find a solution what so ever in case classical methods fail to find any. It is usually achieved by reducing demands on the quality of the solution.

Meta-heuristics. Meta-heuristics have a more specific definition. We will quote Hansen, Mladenović et al. [12]: *"Meta-heuristics, in their original definition, are solution methods that orchestrate an interaction between local improvement procedures and higher level strategies to create a process capable*

of escaping from local optima and performing a robust search of a solution space. Over time, these methods have also come to include any procedures that employ strategies for overcoming the trap of local optimality in complex solution spaces, especially those procedures that utilize one or more neighborhood structures as a means of defining admissible moves to transition from one solution to another, or to build or destroy solutions in constructive and destructive processes.".

The world of meta-heuristics can be divided into two categories: single-solution meta-heuristics focusing on modifying and improving a single candidate solution and population-based meta-heuristics where multiple solutions evolve concurrently. Representatives of the first group are, e.g., Greedy Randomized Adaptive Search Procedure (GRASP), Tabu Search (TS), Variable Neighborhood Search (VNS), etc. GRASP is a multi-start process of creating and improving solutions returning the best amongst all of the restarts at the end. TB is a deterministic local search strategy storing recently visited solutions into memory (i.e., tabu list) capable of avoiding short-term cycling. VNS is based on the idea of systematically changing a neighborhood structure within a local search. From the population-based group, we only name a few examples: Evolutionary Algorithms, Genetic Algorithm or Ant Colony Optimization. Source of these pieces of information is a paper by Gendreau and Potvin [11], nice overview of meta-heuristics and their application to combinatorial optimization problems, which we surely recommend to a more interested reader.

1.3 Literature review

This section is about the state of the art of solving TDP and GSP. Current trends in the TDP research are well-described in the introductions of recent papers Silva et al. [32] and Kulich, Bront & Přeučil [16]. According to these, it seems there are two main streams in solving TDP in operational research community - the Integer Linear Programming (ILP) approach and the heuristic (or meta-heuristic) approach.

Integer Linear Programming approach. ILP approach aims to find the one and only one optimal solution. State of the art algorithms are Branch&Cut (BC) and Branch&Cut&Price (BCP) developed originally for the Time-Dependent Traveling Salesman Problem, a generalization of the classical TSP which also generalizes the TDP. Exact algorithms such as BC or BCP are however limited to small problem sizes. The best one so far was introduced in Abeledo et al. [1]. Their BCP was capable of solving instances with up to 107 vertices in several hours of computing time.

(Meta)heuristic approach. In contrast to ILP, (meta)heuristic techniques are about developing non-exact algorithms. It means that their main objective

is not to find the optimal solution but rather one that is just close to optimum (or let's say good enough) in reasonable computing time even for large instances. The (meta)heuristic approach is more friendly to any possible application than the ILP. To get a comprehensive overview of these methods, we recommend a book *Handbook of Meta-heuristics* by Hansen, Mladenović et al. [12]. In TDP-solving dominates GRASP introduced by Feo and Mauricio [9], VNS proposed by Hansen and Mladenović [20] and some of their modifications and combinations with other meta-heuristics (e.q. TS).

Successful deployment of GRASP in solving TDP was made by Salehipour et al. [26]. Their method was combining GRASP with VNS and its deterministic variant Variable Neighborhood Decent (VND). This method was overcome firstly by Mladenović et al. [19] and their General VNS and than by Silva et al. [32] and their simple multi-start heuristics combined with an Iterated Local Search Procedure (basically some form of GRASP). The last method is to the best of our knowledge the most successful in solving TDP.

More distinct approach to the issue had Kulich, Bront & Přeučil [16]. They proposed several GRASP-based methods for the GSP (and therefore also for the TDP) for which they had direct specific application in robotics. More strict requirements for the feasible computing time emerge from this fact than any of the previously mentioned researchers were putting on themselves. In the paper they show, that their method called GRASP-F deployed on the TDP is nicely comparable in quality to the best one by Silva et al. Their another method GRASP-Int gives slightly worse results in the matter of quality, it meets the strict computing time requirements on the other hand, and is therefore considered the best for their use. They also ran their methods on the GSP and created the first results of this kind. We will use their GRASP-F and GRASP-Int as reference methods in this work.

1.4 Robotic planning

We have already mentioned some of the applications of combinatorial optimization problems such as TSP, TDP, and GSP. Our point of interest is planning in robotics, more specifically the SRSSK, which is a process of autonomous navigation of a mobile robot in a known environment following the objective of finding some stationary object of interest. A different variant of this problem is when the environment is unknown, which at first sight may be similar to an exploration - another well-known problem in the robotic community. Kulich, Přeučil & Bront [18] show that the objectives of search and exploration are dissimilar. The exploration task aims to find the shortest path that completely covers the environment. In the search task on the other hand by following the optimal route, the robot should gain as much new information about the environment in the shortest time as possible. A trajectory that is optimal in exploration does not necessarily minimize the time to find an object of interest along it. In their previously mentioned

later paper [16], they confront the search problem and show, how it can be partially formulated as the GSP.

A similar approach will be ours for the SRSSK, which was to the best of our knowledge originally formulated by Sarmiento et al. [27], where they also extend the problem from Single to Multi-Robot search. The solution approach in Part II of this thesis will be mostly inspired by papers [27], [18], [16] and the latest Kulich & Přeučil [17].

Last note on the background of SRSSK: possible applications of robot search of any kind have a wide range, from finding a specific piece of art in a museum to search and rescue of injured people inside a building or on a battlefield [27].

Part I

Graph Search Problem

Chapter 2

Problem definition

2.1 Graph Search Problem formulation

Let us formulate a variant of the GSP, that we will attempt to solve by the meta-heuristic approach. This formulation is again based on [16]. Assume a special case of a complete graph $G = (V, E)$ describing an instance of the GSP according to Def 1.2 with $V = \{0, 1, \ldots, n-1\}$, where vertex 0 represents the starting point and $n \equiv |V|$ is the total number of vertices. The required information is the position of some object of interest and the probability p_i for every $i \in V$ is represented by a non-negative weight w_i, which is an approximation of the probability that the object of interest will be found when visiting vertex i. The time to reach vertex i from vertex j (where $i \neq j$) is proportional to the distance $d(i, j)$. Let $\mathcal{X} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ be a Hamiltonian path in G, where $x_0 = 0$. Time to reach vertex k by following path $\mathcal{X}$ is defined as

$$\delta_k(\mathcal{X}) \equiv \sum_{i=1}^{k} d(x_{i-1}, x_i). \tag{2.1}$$

Cost of the route is then defined as

$$\texttt{Cost}(\mathcal{X}) \equiv \sum_{k=1}^{n-1} w_k \delta_k(\mathcal{X}), \tag{2.2}$$

and the objective of the GSP is to find the optimal path that minimizes the cost

$$\mathcal{X}^* = \arg\min_{\mathcal{X}} \texttt{Cost}(\mathcal{X}). \tag{2.3}$$

2.2 Solution requirements

Choice of the formulation beneath and not just the one stated in Def 1.2 is not arbitrary. It is built upon our further intention to solve GSP as a

part of SRSSK to which Part II of this thesis is dedicated to. It is meant to be an effective demonstration of GSP usage in robotic planning and also the reasoning behind our ambitions to improve (or overcome by some alternative) the available reference GSP-solving methods in the matter of speed especially. Although speed is crucial, for the search to be effective some demands on solution quality also need to be maintained. We will set this as our secondary goal in the solution approach having speed improvement as the primal one.

Chapter 3

Solution approach

3.1 Reference methods

The purpose of our work is to improve the current state of GSP-solving while following the motivation described in Sec. 2.2. We will use GRASP-F and GRASP-Int proposed in Kulich, Bront & Přeučil [16] as reference methods and therefore we are providing their quick overview. The following is therefore based mostly on [16] and some general thoughts come from Gendreau and Potvin [11].

GRASP is a single-solution multi-start process. Randomized greedy construction heuristics is applied at each restart to create a new solution, which is then improved through a local descent. This is repeated for a fixed number of iterations, and the best overall solution is returned at the end. GRASP-F and GRASP-Int differ only in the improvement of a current solution, and the process of creating the initial route is the same for both. This is how they work:

1. Greedy heuristics creates an initial path P starting from vertex 0. It assigns the best candidate, according to some specific heuristic function, to the end of P until there are no unassigned vertices left. Two heuristics G_{dist} and G_{ratio} were introduced employing two different heuristic functions that are used to evaluate the quality of every candidate:

   - $G_{dist} : f_{dist}(v, u) = d_{v,u}$,
   - $G_{ratio} : f_{ratio}(v, u) = \dfrac{d_{v,u}}{1 + w_u}$,

   where v is the lastly added vertex to P and u is some unassigned vertex. The notation $d_{v,u} \equiv d(v, u)$ will be used for the rest of this text.

   G_{dist} and G_{ratio} are used alternately iteration by iteration, and they both are implemented as their randomized variants G^{rand}_{dist} and G^{rand}_{ratio}, where not always the best-unassigned vertex is added, but rather just

one randomly picked from a list of restricted candidates.

2. After the initial feasible solution is obtained, it is improved by VND. Operations used are *2-opt*: "take two non-adjacent edges and replace them by to new edges" as shown in Fig. 3.1a, and *swap*: "select two vertices in the tour and interchange them" shown in Fig. 3.1b.

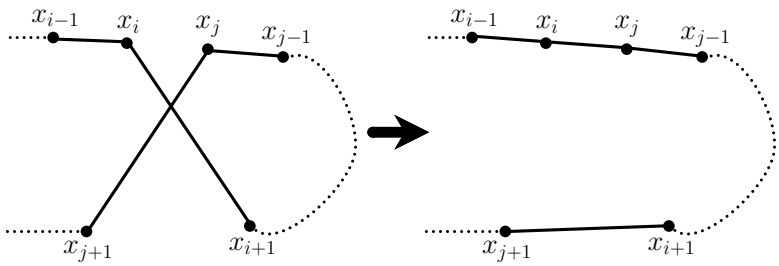
   Equally called improvement methods are derived from these operations. Those methods search the whole neighborhood of the current solution for the best-improving operation which is then is applied. By the neighborhood, we generally mean the set of all possible solutions that might be obtained after some specific pre-defined operation is applied to the current route.

   Improvement methods used are:

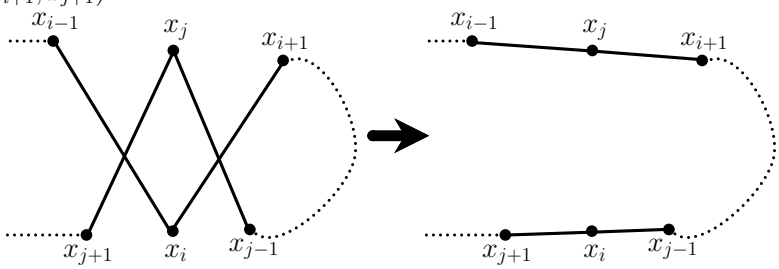
   - ***2-opt***,
   - ***swap***,
   - ***LK-op***,

   where ***2-opt*** and ***swap*** are just mentioned natural derivations of their corresponding operations and ***LK-op*** is a more complex recursive operator with great power to improve but it is also very time-consuming. For the exact definition we will just refer to the original paper [16].

3. The principles described beneath are combined in the GRASP scheme, where the initial solution is created using G^{rand}_{dist} or G^{rand}_{ratio} and then



(a) : *2-opt*: edges (x_i, x_{i+1}) and (x_j, x_{j+1}) removed and replaced by new edges (x_i, x_j) and (x_{i+1}, x_{j+1}).



(b) : *swap*: partitions of the route (x_{i-1}, x_i, x_{i+1}) and (x_{j+1}, x_j, x_{j+1}) removed and replaced by new partitions (x_{i-1}, x_j, x_{i+1}) and (x_{j-1}, x_i, x_{j+1}).

Figure 3.1: Operators *2-opt* and *swap*.

improved in each iteration. The best overall solution obtained after a fixed number of iterations is returned as a result. The improvement phase is done as follows: apply ***2-opt*** and ***swap*** until no further improvements are found and then apply ***LK-op***

- every time in case of GRASP-F,
- only to those routes whose cost is within the 10% of the best solution found so far in case of GRASP-Int.

3.2 Variable Neighborhood Search

Variable Neighborhood Search is based on the idea of improving a single solution exclusively (even by some non-improving steps) rather than creating more solutions and picking the best amongst them. This is achieved by a systematic change of neighborhoods within the search. According to Mladenović and Hansen [20] the VNS is built upon the following perceptions.

1. A local minimum with respect to one neighborhood structure is not necessarily so with another.
2. A global minimum is a local minimum with respect to all possible neighborhood structures.
3. For many problems, local minima with respect to one or several neighborhoods are relatively close to each other.

Let's just for now assume that we have available two neighborhoods derived from the application of operations *2-opt* and *swap*. Neighborhoods can be chosen purposely (*2-opt* and *swap* were) so that Perception 1 applies. Perception 2 must apply, and that can be warranted by a simple thought. If we had an optimal route whose cost is in the global minimum and at the same time there was a neighborhood for which the solution is not a local minimum, then there would be an improving operation that we could apply. But in case we do so, we get a route with a lower cost, and this implies that the original route could not be optimal. This is a contradiction, and therefore the Perception 2 must apply. Last Perception 3 is purely empirical, but it implies that a local optimum might provide some information about the global one.

From the above we could say, that the VNS might be a good try in solving GSP or TDP, especially if the lastly discussed observation 3 was true. Also, practice confirms this idea. VNS is in reality often used in solving combinatorial optimization problems and for a while, it even held primacy in solving TDP[1]. Those are all the reasons we decided to employ VNS in the proposed algorithm.

[1] It was the General VNS by Mladenović et al. [19].

3.3 The algortithm

Let $\boldsymbol{\mathcal{H}}$ be a set of all possible Hamiltonian paths in a complete graph $G = (V, E)$. Now assume a single path $\mathcal{X} \in \boldsymbol{\mathcal{H}}$ and some general operator op which in some pre-defined way changes $\mathcal{X}$ to get a new path $\mathcal{X}' \in \boldsymbol{\mathcal{H}}$. op receives a fixed number p of parameters affecting the character of the change. The parameters have a form of a vector from some set $U \subseteq V^p$ (U must be defined as part of definition of op). op can be non-commutative so in general the order of received parameters matters. Set of all possible parameters op can receive, therefore, must be the set of all possible variations $\boldsymbol{u} = (u_1, \ldots, u_p)$ such that $\boldsymbol{u} \in U$. We define neighborhood $\mathcal{N}_{op}^{\mathcal{X}}$ of path $\mathcal{X}$ as the set of paths obtained by the application of op on $\mathcal{X}$ with every possible variation of parameters op is capable of receiving. We define Local Search Method (LSM) ***op***, a procedure following the scheme shown in Algorithm 3.1. ***op*** systematically searches the $\mathcal{N}_{op}^{\mathcal{X}}$ in order to find it's minimum. We can say, that ***op*** is the LSM derived from the application of op.

In Algorithm 3.2 we show the basic VNS (bVNS) scheme based on Mladenović and Hansen [20]. It starts with creating the initial feasible solution (line 1). Then, for each iteration, two phases follow for at least K times, where K is the number of different operators available. Firstly the best solution so far is changed to get some current working solution. This is done with no demands on improvement, and we call it the Shaking (line 5). Then the working solution is improved by local descent until a local optimum is found. This is called the Local Search (LS) (line 6). Finally, if the cost of the working solution is lower than the cost of the best solution found so far, the working solution becomes the currently best one and the parameter k is set back to 1 (lines 7-9). If not, k is raised by one and the process repeats. If $k = K$ and no better solution is found, the iteration ends. The Shaking is done systematically, where k determines which operator will be applied to get the working solution. The LS is also systematic and is shown in Algorithm 3.3.

Algorithm 3.1: LSM.

Input:
$\mathcal{X}$ - feasible solution
op - an operator

Output:
$\mathcal{X}_{imp}$ - either improved or identical solution to $\mathcal{X}$

```
1 X_imp ← X
2 foreach X' ∈ N_op^X do
3     if Cost(X') < Cost(X_imp) then
4         X_imp ← X'
5 return X_imp
```

Algorithm 3.2: bVNS.

Input:
$\boldsymbol{\mathcal{N}} = \langle op_1, \ldots, op_K \rangle$ - an ordered set of operators
$\boldsymbol{\mathcal{M}} = \langle \boldsymbol{op}_1, \ldots, \boldsymbol{op}_L \rangle$ - an ordered set of LSMs
$iters$ - a number of iterations

Output:
$\mathcal{X}^*$ - the best improved solution

```
1  Create initial solution X_init.
2  X* ← X_init
3  for i = 1, ..., iters do
4      for k = 1, ..., K do
5          Generate path X at random from neighborhood N_{op_k}^{X*}.
6          X_imp ← SearchLocal(M, X)
7          if Cost(X_imp) < Cost(X*) then
8              X* ← X'
9              k ← 1
10 return X*
```

Algorithm 3.3: LS.

```
1  Function SearchLocal(M, X) is
2      l ← 1
3      Loop
4          Get op_l, the l-th LSM from M.
5          Apply op_l on X to get a new path X' ∈ N_{op_l}^{X}.
6          if Cost(X') < Cost(X) then
7              X ← X'
8              l ← 1
9          else
10             if l = L then
11                 End the search.
12                 return X
13             l ← l + 1
```

3.4 Local Search improvement

There is a trouble in the Algorithm 3.1. The cost of every $\mathcal{X}' \in \mathcal{N}_{op}^{\mathcal{X}}$ must be calculated. Route's cost is defined in Eq. 2.2 and it is not an easy calculation.

It has time complexity $\mathcal{O}(n^2)$, and it must be done for $|\mathcal{N}_{op}^{\mathcal{X}}|$ number of times, which slows down the algorithm significantly.

Fortunately, there is a trick that allows us to reduce the time complexity of the cost calculation from $\mathcal{O}(n^2)$ to $\mathcal{O}(1)$ and as a consequence also the time complexity of exploring the entire neighborhood $\mathcal{N}_{op}^{\mathcal{X}}$ from $\mathcal{O}(n^{p+2})$ to $\mathcal{O}(n^p)$. The trick was introduced by Mladenović and Hansen [19]. Let's transfer the Algorithm 3.1 into the Algorithm 3.4, where the function `TestOp` is

$$\texttt{TestOp}(\mathcal{X}, \mathcal{X}') = \texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X}). \tag{3.1}$$

The advantage of this approach is that we don't need to calculate the expression $\texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X})$ directly by computing the costs, but rather express it's value with the help of *op*'s parameters $\boldsymbol{u} = (u_1, \ldots, u_p)$.

Before we show the trick, it's important to note, that it was already implemented in our reference methods GRASP-F and GRASP-Int from [16] on operators *2-opt* and *swap*. Necessary mathematical derivations are shown in the latest paper Kulich & Přeučil [17]. We will use both operators *2-opt* and *swap* (and some others) also in our VNS, and as the essential part of the proposed meta-heuristics, we can't exclude them from this thesis.

First we will show the trick with *2-opt*.

Definition 3.1. Assume the route $\mathcal{X} = \langle x_0, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_{n-1} \rangle$ is given. *2-opt* receives $p = 2$ parameters $(x_i, x_j) \in V^2$ such that $i < j$. Application of *2-opt* on the route $\mathcal{X}$ results in the route

$$\begin{aligned} \mathcal{X}' &= \textit{2-opt}(\mathcal{X}, (x_i, x_j)) \\ &= \langle x_0, x_1, \ldots, x_{i-1}, x_i, x_j, x_{j-1} \ldots, x_{i+1}, x_{j+1}, \ldots, x_{n-1} \rangle. \end{aligned} \tag{3.2}$$

Algorithm 3.4: LSM Improved.

Input:
$\mathcal{X}$ - feasible solution
op - an operator

Output:
$\mathcal{X}_{imp}$ - either improved or identical solution to $\mathcal{X}$

```
1 X_imp ← X
2 improv_best ← 0
3 foreach X' ∈ N_op^X do
4     improv = TestOp(X, X')
5     if improv < improv_best then
6         improv_best ← improv
7         X_imp ← X'
8 return X_imp
```

Proposition 3.2. The improvement obtained by the application of *2-opt* on the route $\mathcal{X}$ (i.e. $\texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X})$) can be computed in the following fashion:

$$\begin{aligned}\Delta_{2\text{-}opt}(\mathcal{X}, i, j) &= \texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X}) \\ &= 2\mathcal{F}_i + (\delta_i + \delta_j + d_{i,j})(\gamma_j - \gamma_i) \\ &\quad + (d_{i,j} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1})(\gamma_{n-1} - \gamma_j) \\ &\quad + 2\mathcal{L}_{j+1} - 2\mathcal{F}_{n-1},\end{aligned} \tag{3.3}$$

where

$$\mathcal{F}_i = \sum_{k=1}^{i} \delta_k w_k, \;\; \mathcal{L}_i = \sum_{k=i}^{n-1} \delta_k w_k \tag{3.4}$$

are contributions to the cost of first and last i vertices respectively. Note, that $\texttt{Cost}(\mathcal{X}) = \mathcal{F}_{n-1} = \mathcal{L}_1$.

$$\gamma_i = \sum_{k=1}^{i} w_k \tag{3.5}$$

is the sum of all weights of the first i vertices.

Proof. For proof, we refer to [17], where the derivation of Eq. 3.3 is shown. □

Assuming that all of $\mathcal{F}$'s, $\mathcal{L}$'s, γ's and also δ's (defined in Eq. 2.1) are precomputed for a given $\mathcal{X}$, the computation of $\Delta_{2\text{-}opt}(\mathcal{X}, i, j)$ can be done in a constant time. Therefore the entire neighborhood $\mathcal{N}^{\mathcal{X}}_{2\text{-}opt}$ can be explored in $\mathcal{O}(n^2)$. In case some better solution is found during the local search procedure, the values previously assumed precomputed are updated at line 7 of Algorithm 3.4. This update can be done iteratively in just $\mathcal{O}(n)$, and therefore the computational complexity of searching the entire neighborhood stays $\mathcal{O}(n^2)$.

Similarly, we can apply the trick on *swap*.

Definition 3.3. Assume the route $\mathcal{X} = \langle x_0, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_{n-1} \rangle$ is given. *swap* receives $p = 2$ parameters $(x_i, x_j) \in V^2$ such that $i < j$. Application of *swap* on the route $\mathcal{X}$ results in the route

$$\begin{aligned}\mathcal{X}' &= swap(\mathcal{X}, (x_i, x_j)) \\ &= \langle x_0, x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_{n-1} \rangle.\end{aligned} \tag{3.6}$$

Proposition 3.4. The improvement obtained by the application of *swap* on the route $\mathcal{X}$ (i.e. $\texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X})$) can be computed in the following fashion:

$$\begin{aligned}\Delta_{swap}(\mathcal{X}, i, j) &= \texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X}) \\ &= w_j \Lambda_1 + (\gamma_{j-1} - \gamma_i)\Lambda_2 + w_i \Lambda_3 + (\gamma_{n-1} - \gamma_j)\Lambda_4 \\ &\quad + (\delta_j - \delta_i)(w_i - w_j),\end{aligned} \tag{3.7}$$

where

$$\begin{aligned}\Lambda_1 &= d_{i-1,j} - d_{i-1,i},\\ \Lambda_2 &= \Lambda_1 + d_{j,i+1} - d_{j,i+1},\\ \Lambda_3 &= \Lambda_2 + d_{j-1,i} - d_{j-1,j},\\ \Lambda_4 &= \Lambda_3 + d_{i,j+1} - d_{j,j+1}.\end{aligned} \tag{3.8}$$

Proof. For proof we refer to [17], where the derivation of Eq. 3.7 is shown. □

The same conclusions as for *2-opt* can be applied for *swap*. The entire $\mathcal{N}^{\mathcal{X}}_{swap}$ neighborhood can be explored in $\mathcal{O}(n^2)$.

3.5 New operators

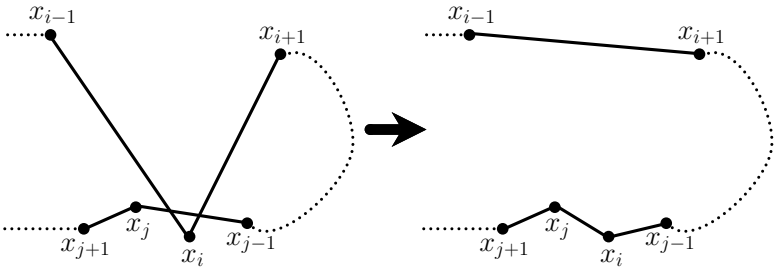
At this moment, we could surely create a meta-heuristics based on the bVNS scheme and operators *2-opt* and *swap* exclusively. Interim results of such method have shown, however, that despite it is much faster than the reference methods, in the matter of quality solution it is dropping behind. The easiest way to remedy its deficiencies would be to raise the number of iterations. Such modification would inevitably increase the computational time, but we want to avoid that. Another way to go would be to drop the bVNS and try some of its extensions or even entirely different meta-heuristics. We decided to stay with bVNS and rather make it more powerful. We will try to achieve that by providing it more neighborhoods for the Shaking and in the LS. The Shaking is crucial as it allows the method to escape local optima and more neighborhoods to choose from raises the chance that this goal is achieved. Better LS can speed up the descent towards those local optima that are close to the global one.

Let us define two new operators displayed in Fig. 3.2:

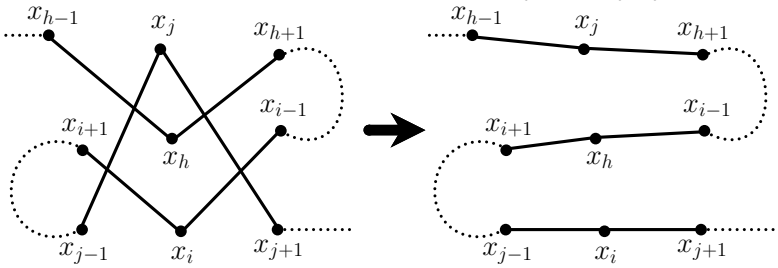
1. *insert*: "Take i-th vertex of the route $\mathcal{X}$ and put it in front of the j-th vertex.",
2. *twist*: "All at the same time put h-th vertex on the position of i-th, i-th on the position of j-th and j-th on the position of h-th in the route $\mathcal{X}$.".

We intend to employ both in the Shaking, but the LS is only reserved for the operation *insert*. *twist* doesn't seem to be a good choice as it takes three vertices as parameters. The act of exploring the entire neighborhood $\mathcal{N}^{\mathcal{X}}_{twist}$ would have time complexity $\mathcal{O}(n^3)$ even after application of the trick introduced in Sec. 3.4. Interim results have shown that the improvement in quality would be insignificant compared to the vastly increased computing time. For *insert* we need to derive its improvement.

Definition 3.5. Assume the route $\mathcal{X} = \langle x_0, x_1, \ldots, x_{n-1}\rangle$ is given. *insert* is non-commutative. For $i = j$ the operation makes no sense. For $i = j - 1$ the vertices already are in the configuration of i-th vertex preceding the



(a) : *insert*: partitions of the route (x_{i-1}, x_i, x_{i+1}) and (x_{j+1}, x_j, x_{j+1}) removed and replaced by new edge (x_{i-1}, x_{i+1}) and new partition $(x_{j-1}, x_i, x_j, x_{j+1})$.



(b) : *twist*: partitions of the route (x_{h-1}, x_h, x_{h+1}), (x_{i-1}, x_i, x_{i+1}) and (x_{j+1}, x_j, x_{j+1}) removed and replaced by new partitions (x_{h-1}, x_j, x_{h+1}), (x_{i-1}, x_h, x_{i+1}) and (x_{j-1}, x_i, x_{j+1}).

Figure 3.2: New operators *insert* and *twist*.

j-th before the operator is applied. We exclude those two options from U. *insert*, therefore, receives $p = 2$ parameters $(x_i, x_j) \in V^2$ such that $i \neq j$ and $i \neq j - 1$. Application of *swap* on the route $\mathcal{X}$ results in the route

$$\mathcal{X}' = \mathit{insert}(\mathcal{X}, (x_i, x_j)) = \begin{cases} \langle x_0, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{j-1}, x_i, x_j, x_{j+1}, \ldots, x_{n-1} \rangle, \\ \qquad \text{if } i < j - 1 \\ \langle x_0, x_1, \ldots, x_{j-1}, x_i, x_j, x_{j+1}, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1} \rangle, \\ \qquad \text{if } i > j \end{cases} . \tag{3.9}$$

Proposition 3.6. The improvement obtained by the application of *insert* on the route $\mathcal{X}$ (i.e. $\texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X})$) can be computed in the following fashion:

$$\Delta_{\mathit{insert}}(\mathcal{X}, i, j) = \texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X}) = \begin{cases} \Lambda_1(\gamma_{j-1} - \gamma_i) + \Lambda_2 w_i + \Lambda_3(\gamma_{n-1} - \gamma_{j-1}) \\ \qquad + (\delta_j - \delta_i) w_i, \text{ if } i < j - 1 \\ (\Lambda_2 - \Lambda_1) w_i + (\Lambda_3 - \Lambda_1)(\gamma_{i-1} - \gamma_{j-1}) \\ \qquad + \Lambda_3(\gamma_{n-1} - \gamma_i) + (\delta_j - \delta_i) w_i, \text{ if } i > j \end{cases} , \tag{3.10}$$

where

$$\begin{aligned} \Lambda_1 &= d_{i-1,i+1} - d_{i-1,i} - d_{i,i+1}, \\ \Lambda_2 &= \Lambda_1 + d_{j-1,i} - d_{j-1,j}, \\ \Lambda_3 &= \Lambda_2 + d_{i,j}. \end{aligned} \tag{3.11}$$

Proof. First we will assume $i < j - 1$. Summing contribution of particular nodes in the order they appear in $\mathcal{X}'$, its cost can be written as:

$$\begin{aligned}
\texttt{Cost}(\mathcal{X}') &= \delta_1 w_1 + \delta_2 w_2 + \cdots + \delta_{i-1} w_{i-1} \\
&\quad + (\delta_{i+1} - d_{i-1,i} - d_{i,i+1} + d_{i-1,i+1}) w_{i+1} \\
&\quad + \cdots + (\delta_{j-1} - d_{i-1,i} - d_{i,i+1} + d_{i-1,i+1}) w_{j-1} \\
&\quad + (\delta_j - d_{i-1,i} - d_{i,i+1} + d_{i-1,i+1} - d_{j-1,j} + d_{j-1,i}) w_i \\
&\quad + (\delta_j - d_{i-1,i} - d_{i,i+1} + d_{i-1,i+1} - d_{j-1,j} + d_{j-1,i} + d_{i,j}) w_j \\
&\quad + \cdots + (\delta_{n-1} - d_{i-1,i} - d_{i,i+1} + d_{i-1,i+1} - d_{j-1,j} + d_{j-1,i} + d_{i,j}) w_{n-1}.
\end{aligned}$$

Using expressions from Eq. 3.11 and also from Eq. 3.4 the cost can be rewritten as:

$$\begin{aligned}
\texttt{Cost}(\mathcal{X}') &= \mathcal{F}_{i-1} \\
&\quad + (\delta_{i+1} + \Lambda_1) w_{i+1} \\
&\quad + \cdots + (\delta_{j-1} + \Lambda_1) w_{j-1} \\
&\quad + (\delta_j + \Lambda_2) w_i \\
&\quad + (\delta_j + \Lambda_3) w_j \\
&\quad + \cdots + (\delta_{n-1} + \Lambda_3) w_{n-1}.
\end{aligned}$$

Improvement obtained by the application of the operation is computed directly as a difference of the costs:

$$\begin{aligned}
\Delta_{insert}(\mathcal{X}, i, j) &= \texttt{Cost}(\mathcal{X}') - \texttt{Cost}(\mathcal{X}) \\
&= \mathcal{F}_{i-1} - \mathcal{F}_{i-1} \\
&\qquad - \delta_i w_i \\
&\qquad + (\delta_{i+1} + \Lambda_1) w_{i+1} - \delta_{i+1} w_{i+1} \\
&\qquad + \cdots + (\delta_{j-1} + \Lambda_1) w_{j-1} - \delta_{j-1} w_{j-1} \\
&\qquad + (\delta_j + \Lambda_2) w_i \\
&\qquad + (\delta_j + \Lambda_3) w_j - \delta_j w_j \\
&\qquad + \cdots + (\delta_{n-1} + \Lambda_3) w_{n-1} - \delta_{n-1} w_{n-1} \\
&= \Lambda_1 w_{i+1} + \cdots + \Lambda_1 w_{j-1} + \Lambda_3 w_j + \cdots + \Lambda_3 w_{n-1} \\
&\qquad + \Lambda_2 w_i + (\delta_j - \delta_i) w_i \\
&= \Lambda_1 (w_{i+1} + \cdots + w_{j-1}) + \Lambda_2 w_i + \Lambda_3 (w_j + \cdots + w_{n-1}) \\
&\qquad + (\delta_j - \delta_i) w_i.
\end{aligned}$$

Using equality from Eq. 3.5 we can alter the result into it's final form:

$$\begin{aligned}
\Delta_{insert}(\mathcal{X}, i, j) = \Lambda_1 (\gamma_{j-1} - \gamma_i) + \Lambda_2 w_i + \Lambda_3 (\gamma_{n-1} - \gamma_{j-1}) \\
+ (\delta_j - \delta_i) w_i.
\end{aligned} \tag{3.12}$$

Now let us assume $i > j$. Using the same process as above we can get

$$\Delta_{insert}(\mathcal{X}, i, j) = \Lambda'_1 w_i + \Lambda'_2 (\gamma_{i-1} - \gamma_{j-1}) + \Lambda'_3 (\gamma_{n-1} - \gamma_i) + (\delta_j - \delta_i) w_i,$$

where

$$\Lambda'_1 = -d_{j-1,j} + d_{j-1,i} = \Lambda_2 - \Lambda_1,$$
$$\Lambda'_2 = \Lambda'_1 + d_{i,j} = \Lambda_3 - \Lambda_1,$$
$$\Lambda'_3 = \Lambda'_2 - d_{i-1,i} - d_{i,i-1} + d_{i-1,i+1} = \Lambda_3.$$

From the above:

$$\begin{aligned}\Delta_{insert}(\mathcal{X}, i, j) = (\Lambda_2 - \Lambda_1)w_i + (\Lambda_3 - \Lambda_1)(\gamma_{i-1} - \gamma_{j-1}) \\ + \Lambda_3(\gamma_{n-1} - \gamma_i) + (\delta_j - \delta_i)w_i.\end{aligned} \tag{3.13}$$

For $i = j$ and $i = j - 1$ *insert* is undefined. Q.E.D. □

3.6 Proposed meta-heuristics

The framework we have built allows us to specify a variant of bVNS by selecting

1. the method that generates the initial solution,
2. the number of iterations $iters$,
3. the ordered set of operators $\boldsymbol{\mathcal{N}}$ used for Shaking, and
4. the ordered set of LSMs $\boldsymbol{\mathcal{M}}$.

The first item of the list above will be inspired by the GRASP-based meta-heuristics proposed by Kulich, Bront & Přeučil [16]. The deterministic variant of the two heuristics G_{dist} and G_{ratio} will be used in general greedy scheme in Algorithm 3.5, where function f is f_{dist} for G_{dist} and f_{ratio} for G_{ratio} defined in Sec. 3.1. Both heuristics create feasible solution and the one with lower cost will be selected as the initial one.

Algorithm 3.5: General greedy scheme.

Input: Set of vertices V, $|V| = n$.

Output: Initial solution - route $\mathcal{X}$.

```
1 X ← ⟨0⟩
2 for k = 1, ..., (n − 1) do
3 |   Let X = ⟨0, x_1, ..., x_k⟩.
4 |   x ← arg min_{v∈V, v∉X} f(x_k, v).
5 |   Append x to the end of X.
6 return X
```

We propose two following variants of bVNS:

1. bVNS-v1 specified by

$$\begin{aligned} iters &= 300, \\ \boldsymbol{\mathcal{N}} &= \{\textit{2-opt}, \textit{swap}\}, \\ \boldsymbol{\mathcal{M}} &= \{\textbf{\textit{2-opt}}, \textbf{\textit{swap}}\}, \text{ and} \end{aligned} \tag{3.14}$$

2. bVNS-v2 specified by

$$\begin{aligned} iters &= 55, \\ \boldsymbol{\mathcal{N}} &= \{\textit{2-opt}, \textit{swap}, \textit{insert}, \textit{twist}\}, \\ \boldsymbol{\mathcal{M}} &= \{\textbf{\textit{2-opt}}, \textbf{\textit{insert}}\}. \end{aligned} \tag{3.15}$$

Chapter 4

Computational results

4.1 Implementation

All algorithms have been implemented in `C++` and compiled by `gcc49`. All experiments of this Part were performed within the same computational environment: a standard PC with an Intel®Xeon®X5560 CPU at 2.80 GHz.

4.2 Reference methods

Computational results for GRASP-based methods are shown in Tabs 4.1 and 4.2. The proposed methods and for comparison also the reference GRASP-F and GRASP-Int were tested on 21 instances from the Traveling Salesman Problem Library (TSPLIB) [25] with sizes between 51 and 1084 vertices. The weights were generated randomly using `random.org` the same way as in [16]. Each vertex was assigned with one of 10000 normally distributed numbers between 1 and 10 in respecting order[1]. Because none of the tested methods is fully deterministic, to obtain reflective results we performed 200 independent runs with different seeds for each experimental setup consisting of an instance and a method.

Before we start any discussion on the results, we will explain the format of the tables as we use the same for all methods. The first two columns show the names of TSPLIB instances as well as their size n explicitly. The third column states the values of Best Known Solution (BKS), which is a cost of the best solution found within this thesis. In other words, if we assume 200 runs on every problem per method (there are four tested methods), it is the best solution found out of the total 800 runs. Some values in this column are shown in **bold** which indicates that the method to which the particular table

[1] That means, e.g., i-th vertex of a TSPLIB instance has assigned i-th random number. The string "2016-09-11" was used as a seed.

refers (further just "the table method") found the best solution in one of its 200 runs[2]. Those values stay numerically the same for every table. Also, we report the percentage gap (%bG) of the best solution found by the table method. The percentage average gap (%aG) follows. The gaps are computed as

$$\%G = 100 \cdot \frac{\text{Sol} - \text{BKS}}{\text{BKS}}, \tag{4.1}$$

where for %bG Sol is the cost of the best solution found by the table method on a particular instance, and for %aG Sol is the average of costs of all solutions found by the table method on the instance. The last three columns give the information about the average computational times. The first value as absolute in seconds (abs [s]). The second value (rel2GF) is relative to GRASP-F, and the last (rel2GI) is relative to GRASP-Int. By the relative, we mean that the value is given by the formula

$$\text{rel2X} = \frac{\text{abs}_\text{X}}{\text{abs}_\text{Sol}}, \tag{4.2}$$

where abs_X is the absolute average time of the corresponding X[3] method and abs_Sol is the absolute average time of the table method. In case of the reference

[2]The Best Known Solution can be found by more methods independently, so in some cases the same value is **bold** for more than one method.

[3]X ≡ "GF" for GRASP-F and X ≡ "GI" for GRASP-Int.

Instance	n	**BKS**	**Solution**		**Time**		
			%bG	%aG	abs [s]	rel2GF	rel2GI
eil51	51	**56795**	0.00	0.38	1.60	1.00	0.57
berlin52	52	**762003**	0.00	0.52	1.01	1.00	0.28
st70	70	102473	0.02	1.35	2.75	1.00	0.21
eil76	76	88890	0.03	0.80	5.78	1.00	0.23
eil101	101	134931	0.66	2.13	13.68	1.00	0.23
bier127	127	23341282	0.67	2.88	19.48	1.00	0.23
ch130	130	1791171	0.53	2.00	19.33	1.00	0.28
d198	198	6702513	0.89	1.89	71.61	1.00	0.46
gil262	262	1488172	1.15	3.37	226.16	1.00	0.19
pr299	299	34512686	1.80	3.67	329.61	1.00	0.20
lin318	318	30964844	2.69	5.52	307.06	1.00	0.27
rd400	400	14904253	2.26	4.57	1209.36	1.00	0.16
pcb442	442	54944343	1.56	3.44	2957.46	1.00	0.09
d493	493	36471199	1.90	3.37	2157.26	1.00	0.18
u574	574	51472970	3.36	4.98	4098.93	1.00	0.15
rat575	575	9755861	2.58	3.78	6825.28	1.00	0.08
d657	657	77687295	2.55	3.84	5580.71	1.00	0.18
u724	724	74984352	1.01	2.83	8879.87	1.00	0.11
rat783	783	17392771	1.97	3.16	25526.71	1.00	0.07
pr1002	1002	633491443	2.36	4.17	40387.28	1.00	0.13
vm1084	1084	522891106	5.00	6.85	44671.12	1.00	0.09
avg	-	-	1.57	3.12	-	1.00	0.21
wavg	-	-	2.35	4.02	-	1.00	0.15

Table 4.1: Computational results for GRASP-F.

methods, those two last columns are not necessary, but we present them to maintain uniformity. For the proposed methods it will be the best sign of eventual speed improvement. The last two rows show the average (**avg**) and the average weighted over the corresponding n (**wavg**) of the values beneath in case the resulting information can be somehow interpretable (otherwise "-").

Now let us take a look at the results for GRASP-F. This particular method found the BKS twice - in case of eil51 and berlin52. %bG is 1.57, and %aG is 3.12 on average amongst all 21 solved problems. GRASP-F is 0.21 times faster (therefore slower) than GRASP-Int. If we emphasize instances with a large number of vertices, we get the weighted average values. The weighted average %bG is 2.35, and %aG is 4.02. GRASP-F is 0.15 times faster than GRASP-Int if we assume the weighted value. From this, we can say that the method's performance gets slightly worse as n gets large. If we focus on the absolute computing times in case of some large instance, e.g., vm1084, we can see it gets long reaching impractical values. In this case, it takes over 12 hours to get a single near-optimal route.

GRASP-Int is doing much better in case of the speed. It is on average 6.42 - 8.37 times faster than GRASP-F. For example, the average computing time in case of vm1084 was reduced from 12 hours to a little bit over 1 hour.

Instance	n	**BKS**	**Solution**				**Time**
			%bG	%aG	abs [s]	rel2GF	rel2GI
eil51	51	56795	0.04	0.42	0.91	1.76	1.00
berlin52	52	**762003**	0.00	0.51	0.28	3.58	1.00
st70	70	102473	0.06	1.75	0.57	4.85	1.00
eil76	76	88890	0.00	1.02	1.33	4.33	1.00
eil101	101	134931	0.37	2.46	3.08	4.44	1.00
bier127	127	23341282	0.82	3.42	4.39	4.44	1.00
ch130	130	1791171	0.17	2.45	5.37	3.60	1.00
d198	198	6702513	0.86	1.93	32.86	2.18	1.00
gil262	262	1488172	1.94	4.15	42.27	5.35	1.00
pr299	299	34512686	1.84	4.09	66.66	4.95	1.00
lin318	318	30964844	3.47	6.04	84.10	3.65	1.00
rd400	400	14904253	2.55	5.17	188.21	6.43	1.00
pcb442	442	54944343	0.98	4.05	257.52	11.48	1.00
d493	493	36471199	2.18	3.94	387.05	5.57	1.00
u574	574	51472970	3.78	5.65	602.39	6.80	1.00
rat575	575	9755861	2.92	4.57	521.98	13.08	1.00
d657	657	77687295	2.17	4.46	996.02	5.60	1.00
u724	724	74984352	2.20	3.73	954.45	9.30	1.00
rat783	783	17392771	2.16	3.93	1794.12	14.23	1.00
pr1002	1002	633491443	1.56	4.65	5184.89	7.79	1.00
vm1084	1084	522891106	4.73	8.11	3921.13	11.39	1.00
avg	-	-	1.66	3.64	-	6.42	1.00
wavg	-	-	2.41	4.71	-	8.37	1.00

Table 4.2: Computational results for GRASP-Int.

The speed gain is balanced by just slightly worse results in the matter of quality. GRASP-Int found the BKS only once in case of berlin52. We can say GRASP-Int gives almost as good results as GRASP-F but is much faster. This observation agrees with conclusions in Kulich, Bront & Přeučil [16].

4.3 Proposed methods

Computational results for bVNS-v1 are shown in Tab. 4.3. It is worse compared to GRASP-based methods in the matter of solution quality. For example, the average value of the %bG is 2.55 for bVNS-v1 while for GRASP-F it is just 1.57 and the average value of the %aG is 5.00 for bVNS-v1 while for GRASP-F it is 3.12. bVNS-v1, on the other hand, appears to be much faster. It is about 127-times faster than GRASP-F and 17-times faster than GRASP-Int on average. For larger instances, the improvement in speed is even more significant as we can see from the weighted average values. E.g., on pr1002 bVNS-v1 is 305-times faster than GRASP-F and 39-times faster than GRASP-Int.

The more powerful version of our original method called bVNS-v2 has its computational results shown in table 4.4. We can notice two significant things

Instance	n	**BKS**	**Solution**				**Time**
			%bG	%aG	abs [s]	rel2GF	rel2GI
eil51	51	**56795**	0.00	1.67	0.08	20.46	11.64
berlin52	52	**762003**	0.00	0.23	0.06	16.61	4.64
st70	70	**102473**	0.00	2.86	0.13	20.45	4.22
eil76	76	88890	0.50	1.07	0.19	29.77	6.87
eil101	101	134931	0.65	4.82	0.37	37.39	8.43
bier127	127	23341282	1.72	4.98	0.66	29.65	6.68
ch130	130	1791171	0.43	3.35	0.62	31.22	8.67
d198	198	6702513	0.38	1.48	1.61	44.60	20.46
gil262	262	1488172	3.30	6.15	3.53	64.11	11.98
pr299	299	34512686	2.61	5.37	4.17	79.08	15.99
lin318	318	30964844	2.96	7.59	5.52	55.64	15.24
rd400	400	14904253	4.09	6.15	9.50	127.37	19.82
pcb442	442	54944343	4.08	5.46	12.17	242.92	21.15
d493	493	36471199	4.21	6.71	22.57	95.59	17.15
u574	574	51472970	3.32	6.50	23.57	173.89	25.56
rat575	575	9755861	4.02	5.80	30.61	222.99	17.05
d657	657	77687295	3.13	6.21	33.95	164.36	29.33
u724	724	74984352	4.48	7.35	43.68	203.28	21.85
rat783	783	17392771	3.14	5.61	58.51	436.31	30.67
pr1002	1002	633491443	5.44	7.19	132.57	304.66	39.11
vm1084	1084	522891106	5.15	8.36	162.22	275.37	24.17
avg	-	-	2.55	5.00	-	127.41	17.18
wavg	-	-	3.75	6.32	-	204.11	23.28

Table 4.3: Computational results for bVNS-v1.

about that table. Firstly - this method is the fastest of all four tested. It solves every problem with (1) $n \leq 130$ within half a second, (2) $n \leq 400$ within 10 seconds and (3) $n \approx 1000$ in about 2 minutes. The same for GRASP-Int is (1) about 5 seconds, (2) about 2 minutes and (3) about 1 hour. And for GRASP-F (1) within 20 seconds, (2) about 20 minutes, (3) about 12 hours. Secondly - this method found the BKS for every single problem available, and therefore it has the average %bG equal to 0.00 precisely. GRASP-F had this value from 1 to 5 for all problems with $n > 200$. Both the **avg** and **wavg** values of %aG are for bVNS-v2 about half of what they are for GRASP-F.

Instance	n	**BKS**	**Solution**				**Time**
			%bG	%aG	abs [s]	rel2GF	rel2GI
eil51	51	**56795**	0.00	0.56	0.03	46.82	26.63
berlin52	52	**762003**	0.00	0.05	0.04	28.61	8.00
st70	70	**102473**	0.00	1.36	0.07	36.97	7.63
eil76	76	**88890**	0.00	0.49	0.09	65.42	15.11
eil101	101	**134931**	0.00	2.41	0.25	55.44	12.49
bier127	127	**23341282**	0.00	3.10	0.41	47.92	10.80
ch130	130	**1791171**	0.00	1.82	0.48	40.29	11.18
d198	198	**6702513**	0.00	0.34	1.13	63.20	29.00
gil262	262	**1488172**	0.00	2.61	3.03	74.66	13.95
pr299	299	**34512686**	0.00	1.57	4.02	81.91	16.56
lin318	318	**30964844**	0.00	3.12	4.29	71.53	19.59
rd400	400	**14904253**	0.00	2.97	9.74	124.12	19.32
pcb442	442	**54944343**	0.00	1.94	12.21	242.25	21.09
d493	493	**36471199**	0.00	1.71	16.24	132.82	23.83
u574	574	**51472970**	0.00	2.60	21.44	191.23	28.10
rat575	575	**9755861**	0.00	1.22	24.16	282.52	21.61
d657	657	**77687295**	0.00	2.34	33.14	168.40	30.06
u724	724	**74984352**	0.00	1.98	45.84	193.71	20.82
rat783	783	**17392771**	0.00	1.77	62.91	405.78	28.52
pr1002	1002	**633491443**	0.00	2.09	117.04	345.08	44.30
vm1084	1084	**522891106**	0.00	2.75	149.34	299.12	26.26
avg	-	-	0.00	1.85	-	142.75	20.71
wavg	-	-	0.00	2.11	-	218.66	25.60

Table 4.4: Computational results for bVNS-v2.

Part II

Single Robot Search for a Stationary Object in a Known Environment

Chapter 5

Problem definition

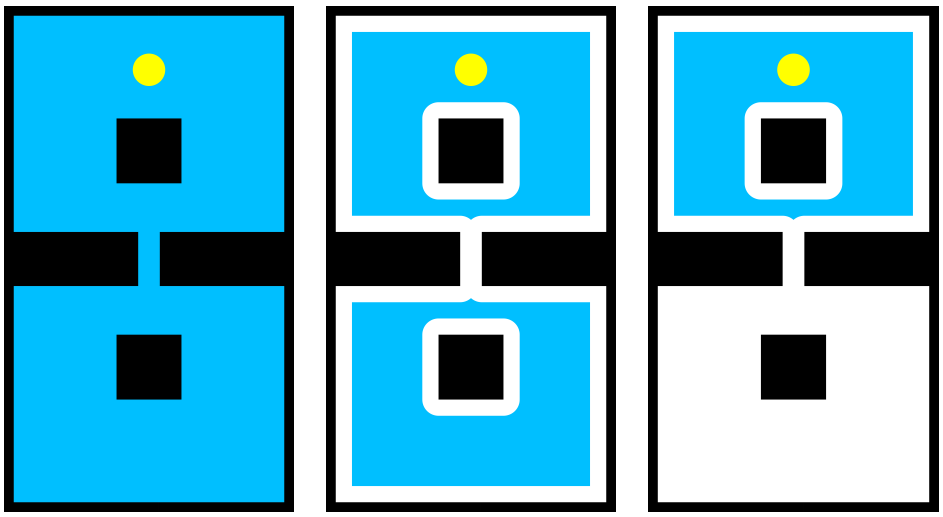
The definition of SRSSK is inspired by [16] and [27], but some of the ideas are also ours. Assume an autonomous mobile robot equipped with a ranging sensor with a fixed, limited range $R > 0$ and 360° field of view operating in a completely known environment. Both the robot and the environment are modeled in 2D. The robot is circular-shaped with radius $r > 0$ and with the sensor placed in its center point. We define robot's position as the coordinates of its center point. The environment W is modeled as a polygon that may contain polygonal holes (obstacles). The obstacles and outer border of W generate both motion and visibility constraints for the robot (the robot can neither go nor see outside of W).

In order to plan a path for a robot with a circular shape, we need to introduce the following formalism. Because $r \neq 0$ (the robot is not modeled as a single point) the robot can only get as close to a border or an obstacle as to the maximal distance of r. We define the deflated environment $\boldsymbol{\mathcal{W}}$, a set of subsets of W, given by the result of offsetting [6] environment W by value $-r$. We assume that the initial position of robot s is given inside one member of $\boldsymbol{\mathcal{W}}$. We define the reachable area $\mathcal{W} \in \boldsymbol{\mathcal{W}}$ for which applies, that $s \in \mathcal{W}$. The difference between W, $\boldsymbol{\mathcal{W}}$ and $\mathcal{W}$ we show in Fig. 5.1.

The search problem is defined as the process of navigation of the robot through the reachable area $\mathcal{W}$ of the environment W ($\mathcal{W} \subseteq W$), in order to find the stationary object of interest placed in the environment randomly on a location $l \in \mathcal{W}$. By finding the object, we understand the situation when it is firstly detected by robot's sensor. Furthermore, we assume that the probability of the object of interest being in any specific point is uniformly distributed throughout the $\mathcal{W}$'s interior and equal to 0 in $W \setminus \mathcal{W}$ (and outside of W, of course). Therefore, the probability of the object being in any subset $A \subseteq \mathcal{W}$ is proportional to the area of A.

Assuming the robot follows the trajectory $\mathcal{R}$ the expected (mean) time T the object is found is given by

$$\mathbb{E}(T|\mathcal{R}) = \int_0^\infty tp(t)dt, \tag{5.1}$$



(a) : The environment W shown in blue.

(b) : Deflated environment $\boldsymbol{\mathcal{W}}$ shown in blue.

(c) : Reachable area $\mathcal{W}$ shown in blue.

Figure 5.1: Difference between W, $\boldsymbol{\mathcal{W}}$ and $\mathcal{W}$ shown on environment with a narrow corridor of width $w < 2r$. The yellow circle in the pictures is a footprint of the robot standing at its initial position s.

where $p(t)$ is the probability of detecting the object at time t. Sensing as well as planning is in robotics performed in discrete times, therefore Eq. 5.1 can be rewritten as

$$\mathbb{E}(T|\mathcal{R}) = \sum_{0}^{\infty} tp(t), \tag{5.2}$$

where $p(t) = \frac{A_t^{\mathcal{R}}}{A_{total}}$ is the ratio of the area $A_t^{\mathcal{R}}$ newly sensed at time t when the robot follows the trajectory $\mathcal{R}$ and A_{total} which is the area of $\mathcal{W}$. The objective of the SRSSK is to find the trajectory $\mathcal{R}^*$ minimizing Eq. 5.2:

$$\mathcal{R}^* = \arg\min_{\mathcal{R}} \mathbb{E}(T|\mathcal{R}) = \arg\min_{\mathcal{R}} \sum_{0}^{\infty} tA_t^{\mathcal{R}}. \tag{5.3}$$

Chapter 6

Solution approach

6.1 General Framework

The most natural way of describing robot's search is shown in the Algorithm 6.1. First, the set of locations to visit is determined, and an order of their visits is planned (lines 1-2). The robot then navigates itself along the trajectory $\mathcal{R}$[1], and in each moment it loads information from sensor and checks whether the object of interest was detected. If so, the search ends, if not, it continues

[1] By the trajectory we mean the continuous curve by which the robot actually goes along and by the path we mean just an ordered set of locations.

Algorithm 6.1: The process of robot's search.

Input:
$\mathcal{W}$ - a map of the reachable area
s - robot's initial position
R - range of robot's sensor

Output:
T - total time from the start of the simulation to the moment when the object was found

1 Determine a set of locations $L \subseteq \mathcal{W}$ so that each point in $\mathcal{W}$ can be seen from at least one location in L.
2 Plan the path $\mathcal{X}$ through every location in L starting at s.
3 Start the navigation of the robot along the trajectory $\mathcal{R}$ to every location in L in the order given by $\mathcal{X}$, $T \leftarrow 0$.
4 **repeat**
5 | Get the newly sensed area from sensor readings.
6 | $T \leftarrow T + \Delta t$
7 **until** *Object is found.*
8 **return** T

Algorithm 6.2: Off-line planning and simulation.

Input:
W - a map of the environment
r - robot's radius
s - robot's initial position
R - range of robot's sensor

Output:
T_f - expected mean time the object is found

1 Determine reachable area $\mathcal{W}$ of environment W.
2 Determine a set of locations $L \subseteq \mathcal{W}$ so that each point in $\mathcal{W}$ can be seen from at least one location in L.
3 Plan the path $\mathcal{X}$ through every location in $L \cup s$ starting at s.
4 $t \leftarrow 0$, $T_f \leftarrow 0$
5 **foreach** g *out of* $\mathcal{X}$ **do**
6 Start the navigation of the robot from its current position to g along the trajectory $\mathcal{R}$.
7 **repeat**
8 $t \leftarrow t + \Delta t$
9 Get the newly sensed area $A_t^{\mathcal{R}}$ from sensor readings.
10 $T_f \leftarrow T_f + t\frac{A_t^{\mathcal{R}}}{A_{total}}$
11 **until** g *is reached.*
12 **return** T_f

to follow $\mathcal{R}$ on next cycle (lines 3-7). At line 6 we indicate, that the total time of simulation T is being recorded - Δt is the difference between a current time and the time of the previous cycle.

In our work, we will rather use an extended search process shown in the Algorithm 6.2. It takes the environment W and robot's radius r as the input and deals with determining the reachable area $\mathcal{W}$. Also, the search is performed on the entire $\mathcal{W}$, without the object of interest placed anywhere for real. The output is expected mean time T_f the object is found, which is more convenient as it is also a quantity to minimize in SRSSK as stated in Chap. 5. In the scheme at lines 1-3 a process called the off-line planning is performed and at lines 4-11 it is the simulation itself.

The rest of this Chapter will inform about our approach to the off-line planning. We will explain and show principles of off-line planning used later in the simulation. We will demonstrate everything on the same instance of the problem. In Fig. 6.1 we show the environment we use for the demonstrations.

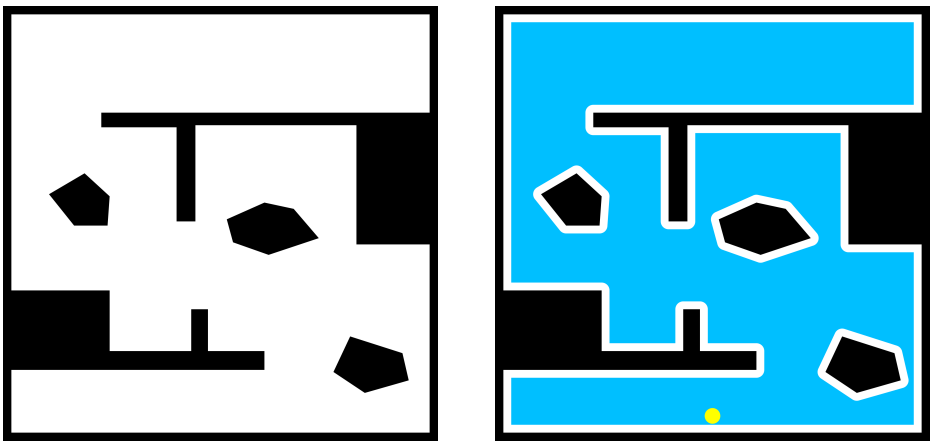


Figure 6.1: Map *complex2*. Left: plane environment W shown in black and white. Right: reachable area $\mathcal{W}$ shown in blue. The yellow circle is robot's footprint.

6.2 Offsetting the environment

For offsetting the environment by the value $-r$ in order to determine the reachable area $\mathcal{W}$ (as well as for some other operations later) we use *Clipper* [13] - a `C++` library which performs line and polygon clipping (intersection, union, difference and exclusive-or) and line and polygon offsetting. The library is based on Vatti's clipping algorithm [34] and it is freeware for both open source and commercial applications. *Clipper* offers more variants of offsetting polygons. They differ in the way they deal with edges of a "blown" polygon. The behavior is given by the joint type used. Two joint types come in our consideration - *round* and *square* - their use we demonstrate on *complex2* map in Fig. 6.2

Figure 6.2: $\mathcal{W}$ generated with both *round* and *square* joint type. The first one is drawn in lighter shade of blue overlapped by the second one in richer color. Left: the whole picture. Right: detail.

The difference between the two is noticeable in the detailed picture. For joint type *round* the inner edges of $\mathcal{W}$ seem to be perfect curves while for joint type *square* they are squared. The first option is the optimal one because it represents exactly what $\mathcal{W}$ is supposed to be - the set of all possible robot's positions in the environment. The second option, on the other hand, is just an approximation of that set with such a welcoming property, that it doesn't make $\mathcal{W}$ too complex compared to W. The decision over which one is better for our use we shall make in next section.

6.3 Triangulation

Line 2 of the Algorithm 6.2 requires to determine a set of locations $L \subseteq \mathcal{W}$ so that each point can be seen from at least one location in L. In this task we must consider all visibility constraints put on the robot - both outer and inner ("blown" obstacles) borders of $\mathcal{W}$ and the range of robot's sensor R. There are several criteria for determining the quality of L. For example, we might try to minimize the number of locations $|L|$. This task is in the literature called the Art Gallery Problem (AGP) [22]. Another way to go would be to determine locations along the shortest path that covers the whole $\mathcal{W}$ - The Shortest Watchman Path Problem [29]. For simplicity, we will not put any similar requirements on set L and rather settle with just the indispensable. We will perform a constrained conforming Delaunay triangulation [23] on reachable area W which creates a triangular mesh of desired properties. Set L will be filled with centroids of all the generated triangles. *Triangle* [31] - a two-dimensional quality mesh generator and Delaunay triangulator [30] - is a `C` tool and library capable of doing just that. We run the triangulation with switch options `pa{`R`}`, where `p` is used to perform the triangulation and

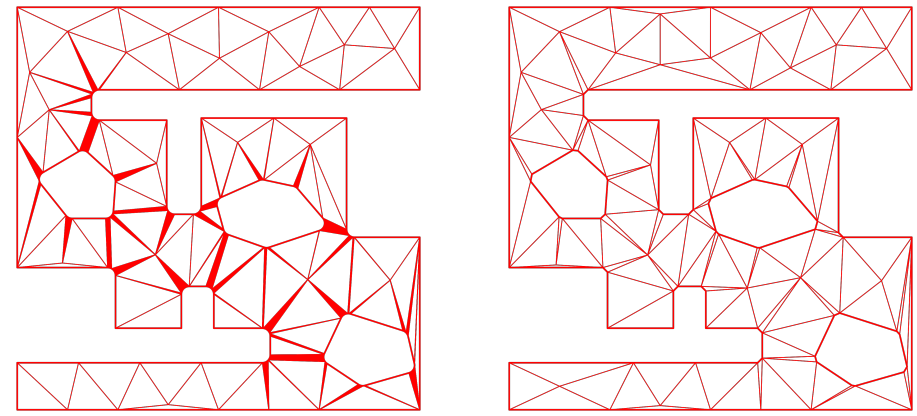


Figure 6.3: Triangulation. Left: performed on $\mathcal{W}$ which was generated using joint type *round*. The number of triangles is 2519. Right: performed on $\mathcal{W}$ which was generated using joint type *square*. The number of triangles is 127.

`a` imposes a maximum triangle area constraint. `{`R`}` stands for its numeric value - range of robot's sensor R is used.

In Fig. 6.3 we show the triangulation performed on the reachable area $\mathcal{W}$ of *complex2* generated with joint type *round* in the first case and *square* in the second. There are some seemingly filled areas in the left picture. Those are, in fact, just large numbers of triangles stacked one on another. This is a good illustration of how much the result of the triangulation is dependent on the complexity of $\mathcal{W}$. Natural condition is to have a small size of L, and the tiny narrow triangles clearly do not add any value to the result. Therefore usage of joint type *round* in offsetting the environment (discussed in the previous section) is impractical and *square* will be preferred. By this choice, we reduced the size of L from 2519 to 127 in case of *complex2*.

6.4 Vertices Reduction

Despite the fact, that minimal size of L is not a strict requirement, it is obvious, that determining L by triangulation results in much bigger $|L|$ than it is necessary for our purpose. Large $|L|$ means greater computational burden and possibly also worse result of the experiment. Therefore we introduce vertices reduction method that removes unnecessary points from L while its desired properties remain. Pseudo-code of that method we show in the Algorithm 6.3 and some parts of the reduction process for *complex2* are displayed in Fig. 6.4.

At lines 1-4 of the Algorithm an initialization is performed. L_r is first assigned with $L \cup s$ and a group of triangles $\boldsymbol{\mathcal{T}_{c_i}}$ is defined for every $c_i \in L_r$. Then the reduction starts. For each $c_i \in L_r$ (line 5) that was not yet removed (line 6) a circle C of radius R with center in c_i is created (line 7). Let us call c_i the reducing point.

Let set L_r be $\{p_0, p_1, \ldots, p_m\}$ and then for every $p_j \in L_r$ ask the following question (lines 8-10). Is it true, that $c_i \neq p_j$ and all triangles that belongs to p_j (elements of $\boldsymbol{\mathcal{T}_{p_j}}$) are both fully inside C and fully visible from c_i inside of $\mathcal{W}$? If the answer is yes, p_j is removed from L_r (line 11) and all triangles originally belonging to p_j becomes the belongings of c_i (line 12). If the answer is no, p_j can not be removed from L_r, as its visit provides some information that the visit of c_i does not. After going through every $p_j \in L_r$ and either removing it or not, c_{i+1} (if not yet removed) becomes the next reducing point and so on. With this procedure we can reduce the original 128 points of $L \cup s$ to just 36 of $L_r \cup s$ on *complex2*.

For the act of determining, whether some points are visible from another point inside environment $\mathcal{W}$ or not, we calculate a visibility graph using *VisiLibity1* [21], a free open source `C++` library for 2D floating-point visibility algorithms, path planning, and supporting data types.

Algorithm 6.3: Vertices reduction procedure.

Input:
$L = \{c_1, \ldots, c_n\}$ - original set of locations
$\boldsymbol{\mathcal{T}_0} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ - set of triangles genereted by the triangulation, where c_i is a centroid of $\mathcal{T}_i$ for every $i \in \{1, \ldots, n\}$
$\mathcal{W}$ - reachable area
s - robot's inital position
R - range of robot's sensor

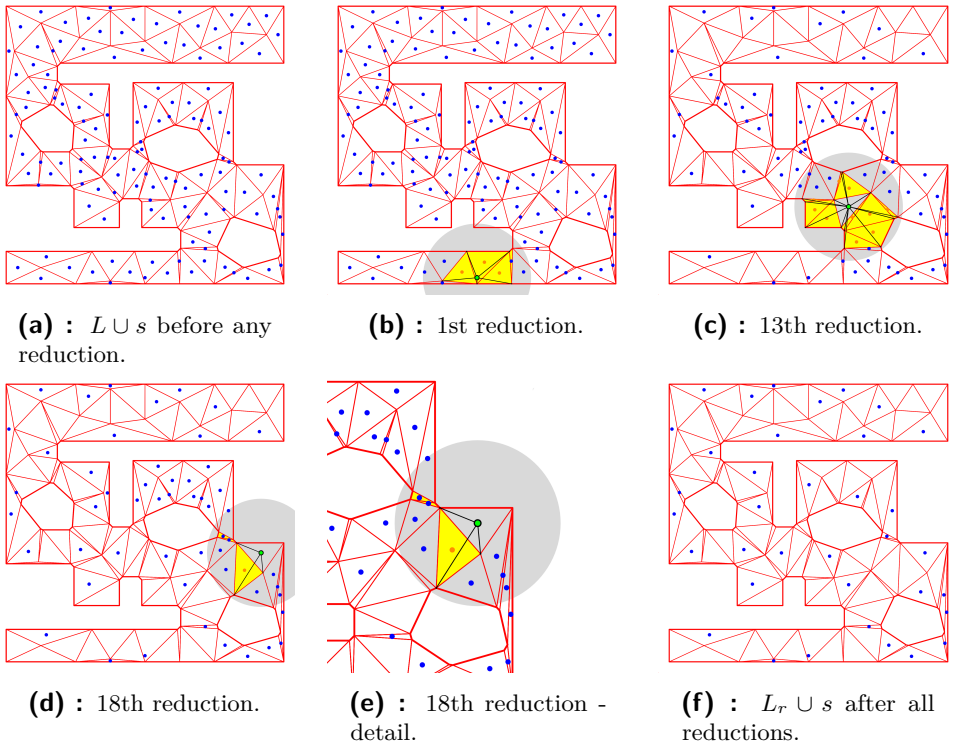
Output:
L_r - reduced set of locations

```
1  L_r ← {c_0 ≡ s, c_1, ..., c_n}
2  T_{c_0} ← ∅
3  for i = 1, ..., n do
4      T_{c_i} ← {T_i}
5  for i = 0, ..., n do
6      if c_i ∈ L_r then
7          Create circle C of radius R with center in c_i.
8          Let L_r ≡ {p_0, p_1, ..., p_m}.
9          for j = 0, ..., m do
10             if c_i ≠ p_j and all memebers of T_{p_j} are both fully inside C
                and fully visible from c_i inside of W then
11                 L_r ← L_r \ p_j
12                 T_{c_i} ← T_{c_i} ∪ T_{p_j}
13 return L_r
```

To make the Algorithm 6.2 still valid with the process of reduction included, we assume that it is a part of line 2 and that L is assigned with L_r after the reduction has finished.

6.5 Path Planning

We are now getting to line 3 of the Algorithm 6.2. Having the set of locations L our objective is to plan a path through every point in L starting at s. This task may be transformed into the GSP defined in Sec. 2.1. An instance of the GSP is described by a complete graph $G = (V, E)$, where V is a set of vertices and E is a set of edges between the vertices. Let us assume $|V| = |L|$ and each point in L corresponds to only just one vertex in V. A distance function $d(i, j)$ we define as the shortest distance the robot can travel to get from i-th to j-th point in L while respecting motion constraints of $\mathcal{W}$. In our implementation, the distance function is in fact represented by a distance matrix D. For calculating D, we use Johnson's algorithm [8] from *The Boost*



(a) : $L \cup s$ before any reduction.

(b) : 1st reduction.

(c) : 13th reduction.

(d) : 18th reduction.

(e) : 18th reduction - detail.

(f) : $L_r \cup s$ after all reductions.

Figure 6.4: Reduction of set $L \cup s$ from the original size 128 to 36. Reachable area $\mathcal{W}$ with triangular mesh is shown in red. Points not yet reduced are blue. Reducing point c_i is green with a black ring and circle C is gray. Groups of triangles entirely inside C are filled with yellow. From the corners of those that are also fully visible go black beams to c_i which symbolizes visibility. Points to-be-removed are orange.

C++ Libraries [28], which is able to find the shortest distance between every pair of vertices in graph G_{boost}. We fill G_{boost} with both the locations from L and the points representing corners of the reachable area $\mathcal{W}$, and we add an edge between every pair a, b of this set for which applies that a is visible from b (and vice versa). We again use *VisiLibity1* library for determining visibility between the two points.

Let us just for now assume, that we also have a corresponding weight of each location in L. The GSP instance is fully defined, and we can solve it using approaches of Part I. As a result, we get the path $\mathcal{X}$ and the off-line planning is finished. Robot's search described in the Algorithm 6.2 continues with the simulation itself and its results we show in next Chapter 7. Next section of this Chapter goes deep into the weights calculation.

6.6 Weights Calculation

According to GSP formulation in Sec 2.1, the weight of each vertex p_i should be an approximation of the probability that the object will be found when

visiting that vertex. In other words, it is the approximation of the amount of information gained (or area newly searched) when visiting p_i.

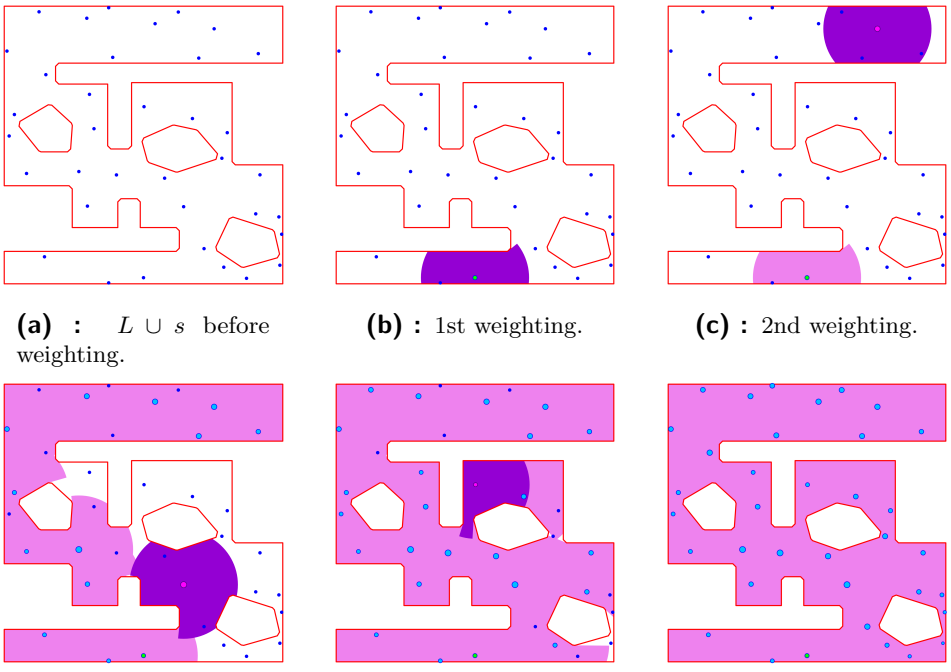
Assuming there is no reduction on set L after the triangulation, the first idea might be to take the area $A_{\mathcal{T}_i}$ of a triangle generated by triangulation that has centroid c_i. The weight of a vertex p_i is then

$$w(p_i) = \frac{A_{\mathcal{T}_i}}{A_{total}}. \tag{6.1}$$

In fact this kind of calculation is far from good approximation of desired probability. Different approach is to create circle C_i of radius R with the center in p_i. Using *Clipper* we get polygon (with holes) $\mathcal{W}_{C_i}$ as the result of intersection of the reachable area $\mathcal{W}$ and the circle C_i. Using *VisiLibity1* we calculate the visibility polygon $P_{\mathcal{W}_{C_i}}$ on $\mathcal{W}_{C_i}$. Weight of vertex i is then

$$w(p_i) = \frac{A_{\mathcal{W}_{C_i}}}{A_{total}}, \tag{6.2}$$

where $A_{\mathcal{W}_{C_i}}$ is the area of $P_{\mathcal{W}_{C_i}}$. This approach works even on reduced $L \cup s$. Visualization of this type of weighting vertices is shown in Fig. 6.5.



(a) : $L \cup s$ before weighting. **(b) :** 1st weighting. **(c) :** 2nd weighting.

(d) : 14th weighting. **(e) :** 25th weighting. **(f) :** $L \cup s$ after weighting.

Figure 6.5: Weights calculation on the reduced set $L \cup s$. Borders of $\mathcal{W}$ are shown in red. Not yet weighted points are blue. Point p_i is pink with blue ring and polygon $P_{\mathcal{W}_{C_i}}$ is violet. Already weighted points are light blue with blue ring, except s having a green center and blue ring. Size of weighted points is given by equation $\text{size}(p_i) = K_1 + K_2 \cdot w(p_i)$, where K_1 and K_2 are some constants. Union of polygons $P_{\mathcal{W}_{C_j}}$ where $j < i$ is pink, and it is in the pictures to show, that by visiting all points from L the robot will truly search 100% of $\mathcal{W}$.

6.7 Proposed variants

We propose the following variants of off-line planning to be tested experimentally in Chapter 7: P00vns, P00gi, P10vns, P10gi[2], P11vns, P11gi. Their name is a code with the following meaning: "P|*weights*|*reduction*|*method*", where *weights* is "0" or "1" if the weights are computed according to Eq. 6.1 or Eq. 6.2 respectively, *reduction* is "1" or "0" if the vertices are reduced or not respectively, *method* is "vns" or "gi" if the method for solving the GSP is bVNS-v2 or GRASP-Int respectively.

[2] A natural question might be, why variants such as P01vns and P01gi are not present. Those variants are combinations of vertices reduction and weighting according to Eq. 6.1. Before we defined the Eq. 6.1, we assumed there is no reduction on set L after the triangulation. If we assumed otherwise, there would be many triangles whose area would not be considered. A workaround to this situation might be to consider in $w(p_i)$ the sum of areas of all the triangles assigned to p_i in the process of reduction. As this option is not straightforward, we decided to leave it out.

Chapter 7

Computational results

7.1 Implementation, simulation, tools

All algorithms have been implemented in `C++`, and the entire project has a form of ROS packages. ROS (Robot Operating System) [24] is a collection of software frameworks for robot software development. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. We use *Kinetic Kame* distribution of ROS. For managing and building the project we use *Catkin Command Line Tools*[1] - a software tool developed by the ROS community designed to efficiently build numerous inter-dependent, but separately developed, CMake projects.

In our code we also use several external libraries (some of them were already mentioned in previous chapter): *Clipper*, *Triangle*, *VisiLibity1*, *The Boost `C++` Libraries* and for graphics and drawing 2D graphics library written in `C`: *Cairo*[2].

The robot used in the simulation is the *TurtleBot* [35] - a low-cost, personal robot kit with open-source software which is based on ROS. Part of the free software coming with *TurtleBot* are several ROS launch files with useful functionality, e.g., teleoperation, navigation and map building. *TurtleBot* kit also allows running a 3D simulation of the robot in a customizable environment without the need to own it physically. The simulation runs in *Gazebo*[3] simulator which can be delivered as a part of ROS.

All experiments of this Part were performed within the same computational environment: a standard notebook with Intel®Core™i5-7300HQ at 2.50 GHz.

[1] Avaliable at `https://catkin-tools.readthedocs.io/en/latest/`.
[2] Avaliable at `https://cairographics.org/`.
[3] Avaliable at `http://gazebosim.org/`.

7.2 Off-line planning

The first set of experiments relate to off-line planning and are based exclusively on our code. We do not deploy ROS simulation in those yet, but we instead use our own robot simulator assuming an ideal behavior of the robot (i.e., it is capable of following the planned trajectory precisely with constant speed, and it can instantly rotate by any angle at the spot).

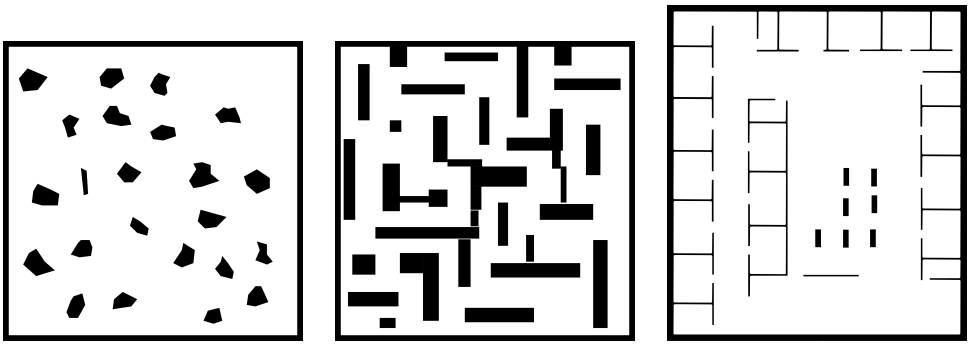
We use 3 environments different from *complex2*, which served only for a demonstration of principles introduced in previous chapter. The maps called *potholes*, *warehouse* and *jari-huge*[4] are shown in Fig 7.1. In our experiments we assume the following properties of the robot:

$$\begin{aligned} r &= 0.3\,m, \\ R &= 3\,m, \\ v &= 0.25\,m/s, \end{aligned} \tag{7.1}$$

where r is its radius, R is the range of its sensor and v is its average speed following any given trajectory $\mathcal{R}$.

We assume an ideal case when the robot always goes by the shortest trajectory $\mathcal{R}_{ideal}$ from its position to the next goal. We tracked the robot as it went along trajectory $\mathcal{R}_{ideal}$ with constant speed v and then we calculated expected mean time of finding the object of interest defined in Eq. 5.2. Results we present in Tab. 7.1. For each map and planning variant, we performed 20 independent runs to provide statistically significant results - numerical values in the table are their means. The first two columns of the tab inform about the experimental setup - map and planning variant. The next four columns have the following meaning: n is the size of $L \cup s$, l is a total length of resulting trajectory $\mathcal{R}_{ideal}$, T_f and T_{f+} are the expected mean times of finding an object with the presumption, that robot starts its journey at time

[4] All used maps are avaliable at `http://agents.fel.cvut.cz/~faigl/planning/maps.xml`.



(a) : *potholes*: 20x20 m **(b) :** *warehouse*: 20x20 m **(c) :** *jari-huge*: 25x28 m

Figure 7.1: The maps used for experiments.

Map	Variant	n	$l\,[m]$	T_f [s]	T_{f+}	Off-line planning times [s] total	red.	vis.	gsp	dis.
potholes	P00vns	376	1048	**205**	247	42.0	0.0	15.6	9.7	15.6
	P00gi	376	765	210	478	268.4	0.0	15.8	236.4	15.5
	P10vns	376	332	248	296	48.0	0.0	15.6	15.9	15.5
	P10gi	376	346	278	622	344.5	0.0	16.6	310.4	16.4
	P11vns	68	162	**205**	**227**	21.5	13.7	5.3	0.3	1.9
	P11gi	68	166	**202**	**230**	28.2	14.9	5.7	5.2	2.0
warehouse	P00vns	272	1731	213	230	16.9	0.0	6.8	7.1	2.6
	P00gi	272	715	210	416	205.8	0.0	6.8	196.1	2.6
	P10vns	272	309	244	263	18.1	0.0	6.8	8.2	2.6
	P10gi	272	332	246	363	117.2	0.0	6.8	107.3	2.6
	P11vns	78	228	**201**	**220**	18.9	14.1	3.5	0.5	0.6
	P11gi	78	219	**206**	**227**	21.6	14.0	3.5	3.2	0.6
jari-huge	P00vns	492	777	571	617	46.4	0.0	19.5	8.2	18.2
	P00gi	492	1139	585	805	219.4	0.0	19.5	181.1	18.1
	P10vns	492	587	487	561	74.0	0.0	19.5	35.0	18.1
	P10gi	492	613	508	1224	716.2	0.0	19.6	677.1	18.1
	P11vns	116	360	**367**	**406**	39.9	28.8	7.3	1.2	2.1
	P11gi	116	378	**381**	**435**	54.5	28.8	7.3	15.9	2.1

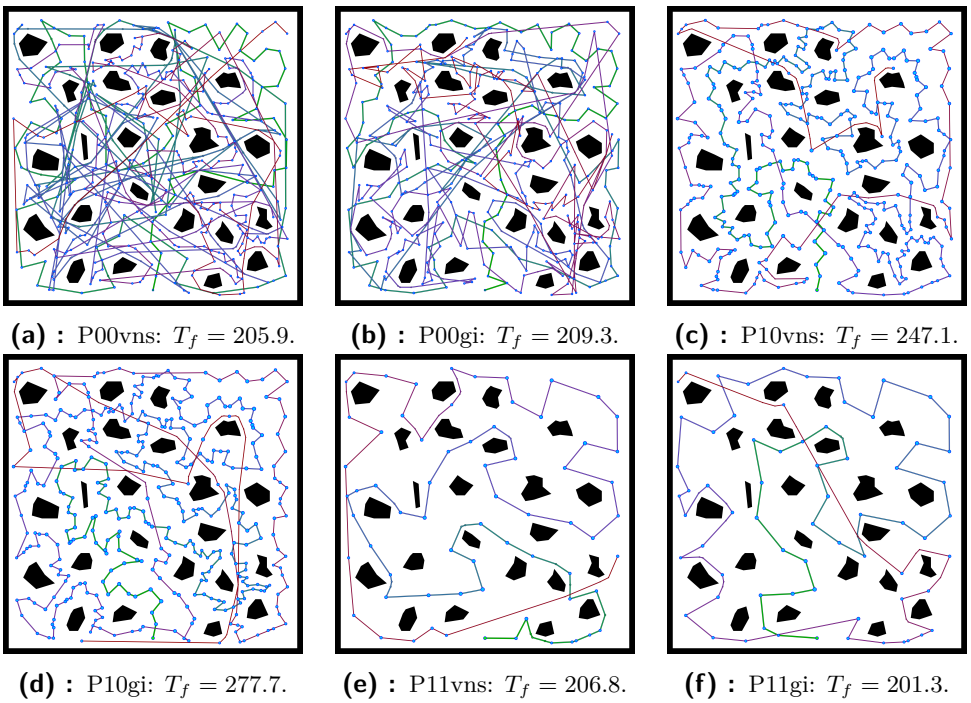
Table 7.1: Off-line planning computational results.

$t_0 = 0$ and t_0 = "total time of off-line planning" respectively. Best average results in these lastly mentioned columns are **bold & underlined** and the second best are just **bold**. The last five columns represent different sections of the planning process and how long they took in seconds - respectively: total time, reduction of vertices, construction of visibility graph, GSP solving and calculation of shortest distances.

The table provides lots of interesting information, and there are many different ways of looking at it. To support the discussion, we show in Fig. 7.2 one resulting trajectory for each planning variant on *potholes* map.

As we can see the overall characteristics of the trajectory depends more on the type of weighting vertices and whether they were reduced or not than on the GSP solving method. Both variants starting with "P00" (further just the P00*) result in trajectories that are hard to follow, and they visually resemble a tangle of wires. Variants starting with "P10" (further just the P10*) on the other hand seem to give much more ordered and logical final trajectories, yet they are sort of unnecessarily wavy. It is worth noting that such a change is caused only by different way of weighting vertices. Finally, we add the reduction of vertices into the process as in case of variants starting with P11 (further just the P11*), and we get very reasonable and the shortest trajectory, which has in fact also the lowest T_f on average. If we compare T_f for P10* and P11* we can with certainty say, that the vertices reduction improved the result.

On the other hand, it is not so easy to make a similar statement about weighting vertices. Resulting expected time for P00* is in case of *potholes*



(a) : P00vns: $T_f = 205.9$. **(b) :** P00gi: $T_f = 209.3$. **(c) :** P10vns: $T_f = 247.1$.

(d) : P10gi: $T_f = 277.7$. **(e) :** P11vns: $T_f = 206.8$. **(f) :** P11gi: $T_f = 201.3$.

Figure 7.2: Six different resulting trajectories $\mathcal{R}_{ideal}$ for six proposed variants on *potholes* map. For each, the value of T_f is close to the average from Tab. 7.1. The trajectory starts at s, and as we go along, it changes color from green to blue and finally to red.

and *warehouse* better than for P10*, in case of *jari-huge* it is the opposite. The question is why. Lets first take P00* which weights vertices according to Eq. 6.1 (triangle areas). The advantage of this approach is that the portion of space $\mathcal{T}_i$ determining a weight of vertex p_i does not overlap with $\mathcal{T}_j$ which applies for every $p_i, p_j \in L$. So there is no redundant information in the weight of vertices. The disadvantage of this approach is that the triangulation itself has just a little to do with robot's view. Despite this fact and also regardless of how the resulting trajectory looks (and how long it is) in our search task it sometimes works surprisingly well. For instance on *potholes* map the resulting T_fs are very close to the best ones produced by P11*.

Determining vertices according to Eq. 6.2 (visibility polygon) is definitely a good idea as P11* shows, but it also has a drawback as we can see on P10*. As we may notice in series of pictures from the process of this kind of weighting (Fig. 6.5), visibility polygons $P_{\mathcal{W}_{C_i}}$ and $P_{\mathcal{W}_{C_j}}$ may overlap for some vertices $p_i, p_j \in L \cup s$. This is a serious problem causing possibly redundant information to be accounted into the party. Performance of P10* is so poor because this happens a lot, as the distribution of vertices in $\mathcal{W}$ is very dense. In case of P11*, this effect is not that strong as the vertices were reduced and visibility polygons overlap just occasionally (but they still do).

Before we start discussing the influence of GSP solving method used in different off-line planning variants, it is important to note, that resulting

expected time T_f (as revealed in previous discussions) depends not only on the GSP method but also on the correctness of vertices weighting, the size of L, etc. Expected time T_f is not the cost of the path defined in Eq. 2.2 and optimized by GSP, even though in ideal case these two values would be directly proportional. But this is not our case, so it is possible to happen that, e.g., P11vns gives worse results of T_f than P11gi despite the fact that in solving the same instance of GSP was bVNS-b2 more successful than GRASP-Int (or otherwise). But now let us compare these two methods anyway by looking at each pair (e.g., 1st and 2nd, 3rd and 4th and so on) of rows individually. In a majority of cases, off-line planning variants using bVNS-v2 performed slightly better with the exceptions of P11* on *potholes* and P00* on *warehouse*.

Benefits of bVNS-v2 over GRASP-Int are more clearly seen on the values of T_{f+}. This type of result assumes that for the entire time the robot is planning its journey it is standing at the place not searching. The value of this time is then accounted into the expected time the object of interest is found. By taking a look at those numbers, it is clear, that bVNS-v2 dominates.

Here are some other insights that can be read from the table. The process of reduction vertices is by itself quite computationally demanding (it takes about 14 seconds on *potholes* and *warehouse* and about 29 seconds on *jari-huge*), but it usually takes down the total time of off-line planning by a big amount (especially in case of GRASP-Int based methods). It shrinks the size of GSP instance by 70-80%, and then every other process takes much less time. Another thing to notice is that GSP instances created by P10* variant take much more time to solve for both bVNS-v2 and GRASP-Int than those produced by P00*. It might be surprising that just the type of weighting vertices alone has such an impact on the difficulty of GSP instance.

7.3 ROS simulation

The last set of experiments that we performed consisted of simulations of the *TurtleBot* in *Gazebo*. The *TurtleBot* has very similar parameters to those we assumed in previous section (Eq. 7.1). To run the simulation, we used unchanged default launch files included with the *TurtleBot*. Each in a different terminal we have run the following commands:

1. `roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=<full path to map.world>`,
2. `roslaunch turtlebot_gazebo amcl_demo.launch map_file:=<full path to map.yaml>`,
3. `roslaunch turtlebot_rviz_launchers view_navigation.launch`.

The first command starts the simulation in *Gazebo*. The second command enables the robot to navigate inside of the simulated world so that it is

capable of receiving navigation goals and planning the route to reach them while avoiding obstacles. Some part of the navigation is also a probabilistic localization system implementing the adaptive Monte Carlo localization approach [10], which uses a particle filter to track the pose of the robot against the known map. Localization is a necessary part of navigation securing robustness. The last command is used to visualize robot's navigation.

The first two commands of the list beneath receive some map file as a parameter. In the first case it is `map.world` which is the map of the desired environment (e.q. *potholes*, *warehouse*, etc.) in a `SDF` format. `SDF`[5] is an `XML` format that describes objects and environments for robot simulators, visualization, and control. It is capable of describing all aspects of robots, static and dynamic objects, lighting, terrain, and even physics. In the second case it is `map.yaml`, which is the known map representation the *TurtleBot* uses for navigation. `YAML` format includes image of the environment, which can be a standard `PNG` file, and then it specifies resolution of the map (meters / pixel), origin (2-D pose of the lower-left pixel in the map) and 2 lightness thresholds defining the state (occupied / unknown / free) of each pixel of the `PNG` image. In order to perform the simulation we had to write a program parsing our representation of the environment (simple `TXT` file specifying scale, robot's starting position and borders with obstacles as sets of points) into formats used by *Gazebo* and the *TurtleBot*: `SDF`, `YAML` and `PNG`.

The *TurtleBot* inside the simulation together with navigation visualization we show in Fig. 7.3.

With everything settled up and running as described beneath our program simulates the search task in the following way. The simulation starts and firstly the off-line planning is performed the same way as in the previous set of experiments. Once the resulting path $\mathcal{X}$ is received, the *TurtleBot* starts executing the search. Coordinates of each point in L are sent to the *TurtleBot* in the order given by $\mathcal{X}$. The program always waits until the robot reaches its current goal and then it sends the next one. The route between every two

[5] More information on `http://sdformat.org/`.

Figure 7.3: *TurtleBot* performing the search inside *Gazebo* simulation on the left. Visualization of robot's navigation in *Rviz*, a 3D visualization tool for ROS, on the right.

Map	Variant	Sim. [s]		Ref. [s]	
		T_f^{sim}	T_{f+}^{sim}	T_f	T_{f+}
potholes	P11vns	224	249	205	**227**
	P11gi	**214**	**244**	**202**	230
warehouse	P11vns	**217**	**237**	**201**	**220**
	P11gi	218	241	206	227
jari-huge	P11vns	**398**	**441**	**367**	**406**
	P11gi	418	477	381	435

Table 7.2: ROS simulation results.

goals is planned by the *TurtleBot* itself. The simulation ends at the moment the last goal is reached. From the point of starting the search to the end of simulation *TurtleBot*'s overall trajectory $\mathcal{R}$ is recorded (robot's coordinates are saved 3-times in a second). The expected times of finding the object of interest T_f^{sim} and T_{f+}^{sim} are calculated after the simulation has finished.

Each experimental setup consisted of the environment (*potholes* / *warehouse* / *jari-huge*) and the off-line planning variant (P11vns / P11gi) and we performed 10 runs for every combination. Results we show in Tab. 7.2. The last two columns contain reference values obtained in the previous set of experiments and extracted from Tab. 7.1. Lower value out of each pair is displayed in **bold**.

P11vns performed the best (compared to P11gi) on the *jari-huge* map. In Fig. 7.4 we show the search progress during time for both P11vns and P11gi inside this environment. In Fig. 7.5 we show typical *TurtleBot*'s trajectory $\mathcal{R}$ inside the *jari-huge* map.

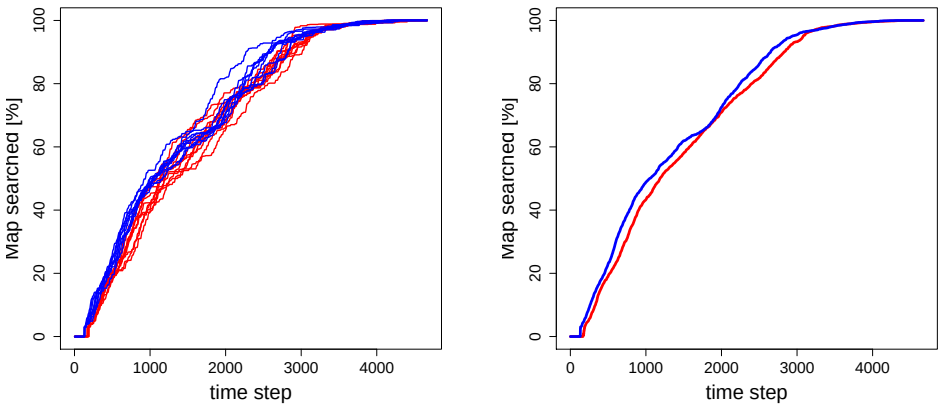


Figure 7.4: The relative amount of searched space during time for the *jari-huge* map. The value is 0 before the search starts (i.e., for the time of off-line planning). Left: progress of all 10 runs of the P11gi (red) and all 10 runs of P11vns (blue). Right: the average values over all 10 runs.

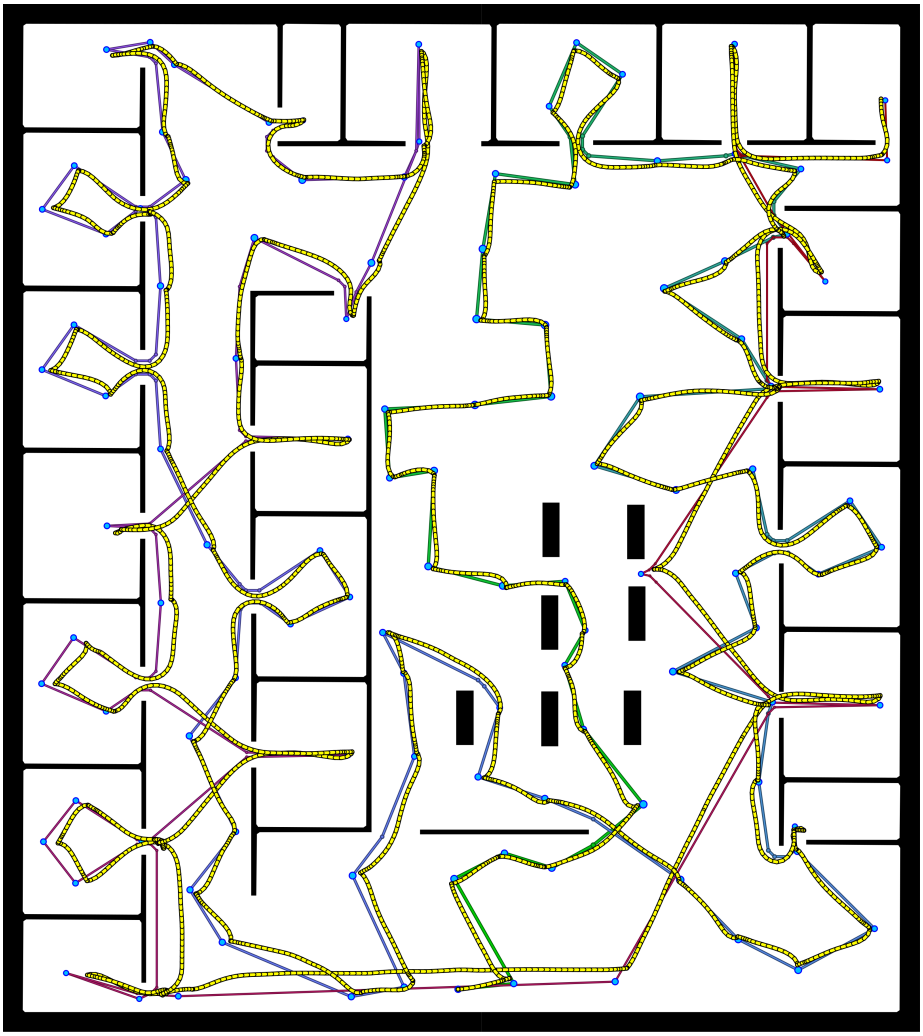


Figure 7.5: Typical *TurtleBot*'s trajectory $\mathcal{R}$ inside *jari-huge* map together with pre-calculated trajectory $\mathcal{R}_{ideal}$. Both trajectories start at s located down on the map in the middle. $\mathcal{R}_{ideal}$ is the underlying one and as we go along it changes color from green to blue and finally to red. $\mathcal{R}$ is the yellow on top.

We can say, that the simulation results are mostly in conformity with the reference. On *potholes* map P11gi is better than P11vns while on *jari-huge* the roles has switched. On *warehouse* P11vns is slightly better, but the difference is so insignificant we can say the two methods performed equally. In Part I of this thesis we have shown, that bVNS-v2 gives on average certainly better results in solving the GSP than GRASP-Int. In our simulation results, this relation is not obvious at all, but this is most likely caused by the other factors affecting the result - the placement of points in L, size of L and the way vertices are weighted. According to expectation, T_{f+}^{sim} values show that bVNS-v2 must be faster then GRASP-Int as it improves a ratio of the two methods results in behalf of P11vns.

Final Remarks

Chapter 8

Conclusions

8.1 Part I

We got acquainted with the GSP and similar combinatorial optimization problems and with different approaches to their solution. We have addressed the GSP with the intention to deploy it in robotic planning. From this evolved strict requirements on our approach in the matter of feasible computational time especially. At the same time, we wanted to maintain the solution quality at the level of our reference methods from Kulich, Bront & Přeučil [16].

We have designed two meta-heuristics for the GSP (and also for the TDP as GSP is its generalization) and implemented them. Both methods were based on the Basic VNS scheme by Mladenović and Hansen [20] systematically changing neighborhoods (which is called the Shaking) within the LS. We have used the same approach to the LS as in [16]. In our first proposed method called bVNS-v1, we incorporated operators *2-opt* and *swap* used in the reference into both the Shaking and the LS. This method was good enough to fulfill our computational time requirements, but in the matter of quality, there was still a room for improvement. We introduced two operators *insert* and *twist* and embedded them into our second proposed method bVNS-v2. This more powerful meta-heuristics used all four mentioned operators in the Shaking while in the LS it employed just *2-opt* and *insert*. For the deployment of *insert* in the LS we had to derive its improvement similarly as it was done in [17] for *2-opt* and *swap*.

Proposed methods together with the reference were tested on 21 instances from the TSPLIB [25] with sizes between 51 and 1084 vertices. The weights were generated randomly. In Tab. 8.1 we show a summary of the obtained results. All values are extracted from the individual tables in Chap. 2 with the only exception of the second column, which is new. $\%_{BKS}$ is the percentage of problems for which the method found the BKS out of all 21 available problems.

Regarding the reference methods, we can say, that GRASP-F is the one

Method	%BKS	Solution %bG avg	%bG wavg	%aG avg	%aG wavg	rel2GF avg	rel2GF wavg	Time rel2GI avg	rel2GI wavg
GRASP-F	9.5	1.57	2.35	3.12	4.02	1.00	1.00	0.21	0.15
GRASP-Int	4.8	1.66	2.41	3.64	4.71	6.42	8.37	1.00	1.00
bVNS-v1	14.3	2.55	3.75	5.00	6.32	127.41	204.11	17.18	23.28
bVNS-v2	100.0	0.00	0.00	1.85	2.11	142.75	218.66	20.71	25.60

Table 8.1: All tested methods performance summary.

that sets a standard for the quality of a solution while GRASP-Int for the computational time. Our proposed method bVNS-v2 overcomes both of these standards as its average %bG, and %aG is for about 1.6 - 2.3 and 1.3 - 1.9 lower than those of GRASP-F respectively and it is an average about 20.7 - 25.6 times faster than GRASP-Int. bVNS-v2 also found BKS for all the tested problems. bVNS-v1 is without surprise not doing so well, but we show its results as a demonstration of how two methods following the same basic scheme can perform differently based on their adjustable settings (i.e., set of operators used in Shaking and LS and number of iterations).

On our way in designing the proposed meta-heuristics, we had to make the following assumption about the GSP: local minima with respect to one or several neighborhoods are relatively close to each other. This was one of the three perceptions on which is build upon successful usage of VNS according to Mladenović and Hansen [20]. Good results of our VNS-based methods show that this assumption may be true for a lot of cases (if not for all) and so we have also learned something about the GSP itself.

8.2 Part II

We employed the proposed method in a robot's search. First, we had to deal with a robot of non-zero radius operating in an arbitrary environment modeled as a polygon with polygonal holes. Offsetting the environment by the negative value of robot's radius made it possible to model the robot by a single point in part of the environment we call the reachable area $\mathcal{W}$. Next, we had to determine a set of locations L such that each point in $\mathcal{W}$ was visible from at least one of these locations. For this, we have used constrained conforming Delaunay triangulation on $\mathcal{W}$ with constraints on a maximal triangle area and we created a triangular mesh. Centroids of the generated triangles were then assigned as the basis for L. We introduced an optional method capable of reducing L to 70-80% of its original size while maintaining its desired properties. We adopted two ways of weighting vertices: one based on the triangle areas and one on the area of visibility polygons. The first one was applicable on the original L only, the second one on both original and reduced L.

We proposed six variants of off-line planning each based on one of the possible combinations of weighting vertices, their optional reduction and GSP-solving method employed (bVNS-v2 / GRASP-Int). We evaluated the variants experimentally in our simple robot simulator. The best variants were those employing vertices reduction and weighting based on the visibility polygons. Solutions generated by bVNS-v2 was on average slightly better, yet the difference between the two GSP methods was not significant. This might be caused by other factors having a greater effect on the result such as the placement of points in L, size of L and the way vertices are weighted. Advantages of bVNS-v2 were mostly in its speed, which made it possible for the robot to finish the off-line planning early and set off sooner than in case of GRASP-Int. With the best couple of variants we performed a simulation on the *TurtleBot* in ROS and results of that simulation were in conformity with previous experiments.

Chapter 9

Future research

Based on our observations we can say, that the VNS can get more powerful by providing it more (and better) operators for the Shaking and the LS. For future research, therefore, we suggest developing several new operators and incorporate them into the VNS, then fine-tune the basic scheme. Interesting might be to extend the VNS by one of its variations from [20], e.g., to Variable Neighborhood Decomposition Search (VNDS) which is a two-level VNS based upon decomposition of a problem, or the Skewed VNS (SVNS), addressing the problem of exploring valleys far from the incumbent solution. Another appealing option is to incorporate VNS into the GRASP-scheme and use advantages of both approaches. Lastly, we mention the possibility to employ TS inside the VNS fully in the LS phase or/and just use its principles (short-memory) to prevent possible cycling or avoid non-promising neighbors in the Shaking or/and in the LS.

Regarding the search problem, in order to obtain better or more useful results we suggest the following list of possible focus areas of our future work:

1. a distinction between robot's reachable area and the area it is truly capable of seeing,
2. better generation of points in L,
3. advanced weighting,
4. introduction of on-line planning,
5. generalization of the GSP by implementing more aspects of the planning in it.

The first item of the list addresses the following issue. We defined the reachable area $\mathcal{W}$ as the set of all points robot's coordinates can attain inside an environment W. This allowed us to model the robot as a single point and we shrank the task of searching inside the entire W to searching just inside $\mathcal{W}$. In fact, for doing that we had no other justification than a simplification of the problem. In Fig. 9.1 we show the reachable area $\mathcal{W}$ together with the area the robot is truly capable of seeing - let us call it $\mathcal{W}^+$. In some

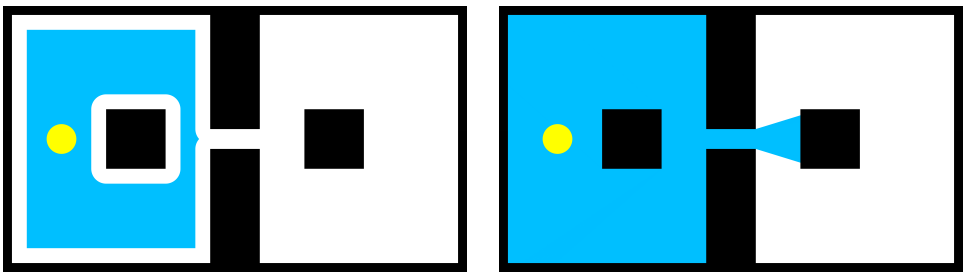


Figure 9.1: Left: reachable area $\mathcal{W}$ in blue. Right: the area the robot is capable of seeing $\mathcal{W}^+$ in blue. The yellow circle is robot's footprint.

environments this area is still not the entire W, yet it can be significantly larger than $\mathcal{W}$. If we assume the object of interest being somewhere inside $\mathcal{W}^+$, the robot certainly can detect it, so there is no reason to constrain the search just to $\mathcal{W}$ rather than to $\mathcal{W}^+$. Our goal is to consider this fact in our future approaches.

The way we determine the set L has a lot of inconveniences too. One of them is that the number of triangles generated by triangulation is strongly dependent not only on the size of the environment but also on its complexity (e.g., if some curve is present). We have shown that on the reachable area $\mathcal{W}$ generated with the usage of joint type *round* (Sec. 6.3). The number of generated points by triangulation is unnecessary large even if the environment is not that complex and its proper reduction (by the introduced method) is too computationally demanding. The best way to tackle this problem would be to solve it as a separate optimization task similar to the AGP. Kazazakis & Argyros [14] address this problem. Their method is based on dividing the non-convex polynomial environment into a set of convex polygons. Amit et al. [2] also have similar approach. We are looking to get inspired by these authors in the future.

We have revealed a drawback of our way of weighting vertices by visibility polygons. In case some of them overlap redundant information is accounted into the weights. This happens even more often if the vertices are densely distributed. An improved version may be based on the following. We take every doublet of vertices and determine their visibility polygons the same way as we already do. We check whether they overlap and in case they do we find out by what amount. We take that amount, divide it by two, and we let the result to be reflected in the weights of both points equally. For even more advanced approximation, we take every triplet, quadruplet and so on.

The idea of on-line planning is established on re-planning during the search. The area that was already seen is at some moment cut off from the rest of the environment, and on the remaining unsearched part, the planning is performed again. The on-line planning may include a new generation of the points in L, their weighting and solving GSP or just the re-weighting and solving GSP on the unvisited subset of the original set L. This approach is particularly interesting, as it makes the usage of slow methods almost

impossible over the fast ones. The true potential of meta-heuristics proposed in this thesis might get revealed.

The last point in our improvement suggestions is to generalize the GSP by implementing more aspects of the planning in it. The fact is that the exact amount of new information gained by visiting each vertex is dependent on the order of vertices in the planned path $\mathcal{X}$. From this, we can conclude that even the most accurate way of weighting vertices is just an approximation of the desired amount of new information as long as it does not account the order of visits. We propose a generalization of the GSP, where the weights of vertices are not pre-defined, but rather calculated for each individual path (e.q., by some external function). To take account of the angles by which the robot must turn in the vertices might also be a useful generalization. The natural condition would be to minimize the total amount of time the robot spends on turning around by some angle while in place. Paths that do not contain lots of sharp angles would then be preferred over those that do.

Surely we could continue with ideas as the world of possibilities is indeed broad, but at some point, we have to stop. In more distant future, we consider extending our range of the problem into the multi-robot case or searching in an environment that is not known apriori. We look forward to whatever the future might bring.

Appendices

Appendix A

List of abbreviations

Abbreviation	Meaning
TSP	Traveling Salesman Problem
TDP	Traveling Deliveryman Problem
GSP	Graph Search Problem
SRSSK	Single Robot Search for a Stationary Object in a Known Environment
GRASP	Greedy Randomized Adaptive Search Procedure
TS	Tabu Search
VNS	Variable Neighborhood Search
ILP	Integer Linear Programming
BC	Branch&Cut
BCP	Branch&Cut&Price
VND	Variable Neighborhood Descent
LSM	Local Search Method
bVNS	basic Variable Neighborhood Search
LS	Local Search
TSPLIB	Traveling Salesman Library
BKS	Best Known Solution
AGP	Art Gallery Problem
ROS	Robot Operating System

Appendix B

Bibliography

[1] Hernán Abeledo, Ricardo Fukasawa, Artur Pessoa, and Eduardo Uchoa. The time dependent traveling salesman problem: polyhedra and algorithm. *Mathematical Programming Computation*, 5(1):27–55, Mar 2013.

[2] Yoav Amit, Joseph S. B. Mitchell, and Eli Packer. Locating guards for visibility coverage of polygons. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 120–134, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[3] Giorgio Ausiello, Stefano Leonardi, and Alberto Marchetti-Spaccamela. On salesmen, repairmen, spiders, and other traveling agents. In Giancarlo Bongiovanni, Rossella Petreschi, and Giorgio Gambosi, editors, *Algorithms and Complexity*, pages 1–16, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[4] Lucio Bianco, Aristide Mingozzi, and Salvatore Ricciardelli. The traveling salesman problem with cumulative costs. *Networks*, 23(2):81–91, 1993.

[5] K. Chaudhuri, B. Godfrey, S. Rao, and K. Talwar. Paths, trees, and minimum latency tours. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 36–45, Oct 2003.

[6] Xiaorui Chen and Sara McMains. Polygon offsetting by computing winding numbers. 01 2005.

[7] William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2011.

[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Johnson's algorithm for sparse graphs. In *Introduction to Algorithms*, pages 636–640. McGraw-Hill Higher Education, 2001.

[9] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, Mar 1995.

[10] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 343–349, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[11] Michel Gendreau and Jean-Yves Potvin. Metaheuristics in combinatorial optimization. *Annals of Operations Research*, 140(1):189–213, Nov 2005.

[12] Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. *Handbook of Metaheuristics*. Springer US, Boston, MA, 2010.

[13] Angus Johnson. Clipper - an open source freeware library for clipping and offsetting lines and polygons. `http://www.angusj.com/delphi/clipper.php`, 2014.

[14] G. D. Kazazakis and A. A. Argyros. Fast positioning of limited-visibility guards for the inspection of 2d workspaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2843–2848 vol.3, 2002.

[15] Elias Koutsoupias, Christos Papadimitriou, and Mihalis Yannakakis. Searching a fixed graph. In Friedhelm Meyer and Burkhard Monien, editors, *Automata, Languages and Programming*, pages 280–289, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[16] Miroslav Kulich, Juan José Miranda Bront, and Libor Přeučil. A metaheuristic based goal-selection strategy for mobile robot search in an unknown environment. *Computers & Operations Research*, 84:178–187, 2017.

[17] Miroslav Kulich and Libor Přeučil. Multi-robot search for a stationary object placed in known environment. *In review*.

[18] Miroslav Kulich, Libor Přeučil, and Juan José Miranda Bront. Single robot search for a stationary object in an unknown environment. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5830–5835, May 2014.

[19] Nenad Mladenović, Dragan Urošević, and Saïd Hanafi. Variable neighborhood search for the travelling deliveryman problem. *4OR*, 11(1):57–73, Mar 2013.

[20] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097 – 1100, 1997.

[21] K. J. Obermeyer and Contributors. VisiLibity: A c++ library for visibility computations in planar polygonal environments. `http://www.VisiLibity.org`, 2008. R-1.

[22] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., New York, NY, USA, 1987.

[23] L. Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4(1):97–108, Jun 1989.

[24] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[25] Gerhard Reinelt. Tsplib - a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[26] Amir Salehipour, Kenneth Sörensen, Peter Goos, and Olli Bräysy. Efficient grasp+vnd and grasp+vns metaheuristics for the traveling repairman problem. *4OR*, 9(2):189–209, Jun 2011.

[27] Alejandro Sarmiento, Rafael Murrieta-Cid, and Seth Hutchinson. A multi-robot strategy for rapidly searching a polygonal environment. In Christian Lemaître, Carlos A. Reyes, and Jesús A. González, editors, *Advances in Artificial Intelligence – IBERAMIA 2004*, pages 484–493, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[28] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[29] T. C. Shermer. Recent results in art galleries [geometry]. *Proceedings of the IEEE*, 80(9):1384–1399, Sep 1992.

[30] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[31] Jonathan Richard Shewchuk. Triangle: A two-dimensional quality mesh generator and delaunay triangulator. `https://www.cs.cmu.edu/~quake/triangle.html`, 2005.

[32] Marcos Melo Silva, Anand Subramanian, Thibaut Vidal, and Luiz Satoru Ochi. A simple and effective metaheuristic for the minimum latency problem. *European Journal of Operational Research*, 221(3):513–520, 2012.

[33] John N. Tsitsiklis. Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3):263–282, 1992.

[34] Bala R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, July 1992.

[35] Melonee Wise and Tully Foote. Turltebot: a low-cost, personal robot kit with open-source software. `https://www.turtlebot.com/`, 2010.

Appendix C

CD Content

File	Description
`F3-BP-2018-Jan-Mikula.pdf`	The `PDF` file containing this thesis.
`thesis.zip`	The `ZIP` archive containing the LaTeX source files of this thesis.
`results.zip`	The `ZIP` archive containing the results presented in this thesis together with the scripts for processing them and creating the LaTeX tables.
`gsp.zip`	The `ZIP` archive containing the source files of Part I of this thesis. Instructions on building the project and running the experiments are included in `README.txt`.
`srssk.zip`	The same as previous but for Part II.