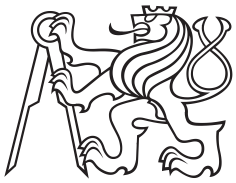
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Search for a static object in a known environment

Bc. Jan Mikula

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and robotics
Subfield: Cybernetics and robotics
January 2021

MASTER'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: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

Search for a static object in a known environment

Master's thesis title in Czech:

Hledání statického objektu ve známém prostředí

Guidelines:

1. Get acquainted with state-of-the-art meta-heuristics for the Traveling Deliveryman Problem (TDP).
2. Design and implement a meta-heuristic for the TDP which considers a limited computational time.
3. Compare experimentally properties of the implemented algorithm with state-of-the-art methods. Describe and discuss the obtained results.
4. Design and realize extensions of the proposed TDP solver for robotic applications.
5. Evaluate experimentally properties of the extended algorithm. Describe and discuss the 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.
[4] Hoos, H.H., Stützle, T., 1998. Evaluating Las Vegas Algorithms - Pitfalls and Remedies. Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence , 238–24.

Name and workplace of master's thesis supervisor:

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

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

Date of master's thesis assignment: **15.09.2020** Deadline for master's thesis submission: **05.01.2021**

Assignment valid until: **19.02.2022**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master'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 master's thesis, the author must state the names of consultants and include a list of references.

. | Student's signature

Date of assignment receipt

Acknowledgements

I want to express my gratitude to my supervisor RNDr. Miroslav Kulich, Ph.D., for his invaluable guidance, advice, patience, and warm approach. I also wish to thank my family and significant others for their endless support during my studies. My other thanks belong to Prof. Marcos Melo Silva, who kindly provided his code and datasets, and Ing. Jan Vidašič for contributing parts of his code and some advice.

Declaration

I hereby declare that I have completed this thesis on my own 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, January 5th, 2020

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 5. 1. 2020

. . . . . .

Abstract

The *mobile search* consists of finding one or several *targets* in a given environment by one or several mobile sensors. We assume a single static object secretly placed inside a known 2D polygonal environment and aim to find the object by a single mobile robot as quickly as possible on average. The robot is equipped with a sensor of 360° view and limited visibility range, sensing is performed throughout the whole search, and the robot can recognize the object once it appears in its field of view. The core of the problem is planning the search efficiently. In this thesis, we design, implement, and evaluate an original framework for the *mobile search*. In general, the problem is solved by a standard decoupling approach; nevertheless, both parts of our solution are innovative. We propose a novel way to discretize the environment utilizing a solution to the related *watchman route problem*. We also introduce a general metaheuristic for finding efficient plans on several discrete models of the problem. The new metaheuristic, which considers limited computing time, is first designed using a general run-time distribution methodology for the most basic model — the *traveling deliveryman problem*. Evaluated on several sets of standard benchmark instances used by the *operations research* community, it significantly outperforms the current best approach from the literature under hard limit settings with limits ranging from 1 to 100 seconds. Still, it provides competitive results in the traditional sense and with cost targets corresponding to the best-known solutions worsened by about 1 %. The metaheuristic is further extended to better model the *mobile search* by considering non-equal and non-static locations' information gains, an effort needed to turn the robot and sensing on the way between locations. Together, the proposed discretization and metaheuristic produce efficient *mobile search* strategies, as shown by our idealized custom simulations and experiments in a realistic robotic simulator. In real life, our solution can be used as an efficient planner for a *search and rescue* scenario in which a mobile robot or other agent searches for victims after some catastrophic event.

Keywords:

mobile robotics
mobile search
metaheuristics
routing problems
traveling deliveryman problem
watchman route problem

Supervisor:

RNDr. Miroslav Kulich, Ph.D.

IMR - Intelligent Mobile Robotics,
CIIRC, CTU in Prague,
Jugoslávských partyzánů 1580/3,
160 00 Praha 6, Dejvice,
Czech Republic

Abstrakt

Problém *mobilního hledání* obecně spočívá v nalezení jednoho nebo více cílů v daném prostředí pomocí jednoho nebo několika mobilních senzorů. My předpokládáme jeden statický objekt, který je, neznámo kam, umístěn dovnitř známého 2D polygonálního prostředí a chceme ho nalézt s pomocí jediného mobilního robotu v průměru co nejrychleji. Robot je vybaven senzorem s 360° rozhledem a omezeným dosahem viditelnosti, který po celou dobu hledání snímá okolí. U robotu se předpokládá schopnost rozpoznat objekt zájmu pokud se vyskytuje v jeho zorném poli. Jádrem problému je tedy naplánovat co nejeefektivnější strategii hledání. V této práci navrhneme a implementujeme novou metodu pro tento problém a experimentálně ověříme její vlastnosti. V obecné rovině problém řešíme standardně a to rozdělením na diskretizaci a optimalizaci. Obě části našeho řešení jsou nicméně inovativní. Navrhujeme nový způsob, jak diskretizovat prostředí s využitím řešení souvisejícího, tzv. *hlídačova problému*. Zavádíme také obecnou metaheuristiku produkující efektivní plány pro několik diskrétních modelů původního problému. Nová metaheuristika, která bere v úvahu omezený výpočetní čas, je nejprve navržena pro nejjednodušší model — *problém cestujícího doručovatele* — a to pomocí obecné metodiky založené na vyhodnocení distribuce výpočetního času z mnoha běhů. Testována na několika sadách standardních instancí používaných komunitou z *operačního výzkumu*, naše metaheuristika výrazně překonává současný nejlepší přístup z literatury v experimentech s omezením na výpočetní čas v rozmezí od 1 do 100 sekund. Dále poskytuje konkurenceschopné výsledky v tradičním smyslu a v experimentech s danou cílovou kvalitou řešení, která odpovídá nejlepšímu známému řešení zhoršenému přibližně o 1 %. Metaheuristika je dále rozšířena tak, aby lépe modelovala problém *mobilního hledání*, a to zohledněním úsilí potřebného k otáčení robota, snímání na cestě mezi lokacemi a dalších reálných aspektů problému. Navrhovaná diskretizace a metaheuristika společně produkují efektivní strategie pro *mobilní hledání*, jak ukazují naše vlastní idealizované simulace a také experimenty v realistickém prostředí robotického simulátoru. V reálném životě lze naše řešení použít například jako efektivní plánovač v krizovém scénáři, kde mobilní robot nebo jiný druh agenta hledá oběti po nějaké katastrofě.

Klíčová slova:

mobilní robotika
mobilní hledání
metaheuristiky
směrovací problémy
problém cestujícího doručovatele
hlídačův problém

Překlad názvu:

Hledání statického objektu ve známém prostředí

Contents

Figures

Tables

Algorithms

Chapter 1

Preliminaries

1.1 Introduction

Target detection and *tracking* play a significant role in many robotic applications, which has led to the design and development of various theoretical approaches and practical solutions to *target*-related robotic problems [1]. *Mobile search*, one of the *target detection* problems, consists of finding one or several *targets* in a given environment by actively sweeping the environment with one or several mobile sensors. The moment of finding the *target* is when it first appears in the sensor's field of view, and the *target* is assumed to be always reliably detected and recognized once this situation occurs. The core of the problem is at planning the search efficiently.

This project's primary goal is to design, implement, and evaluate a framework capable of producing efficient search strategies for the *mobile search* problem. More specifically, we address a variant that assumes a static object of interest (*target*) placed in a priory known 2D polygonal environment, whereas the goal is to find the object by a single mobile robot as quickly as possible on average. Furthermore, we assume the robot is equipped with a sensor of 360° view and limited visibility radius $r_{\mathcal{V}} \in \mathbb{R}^{+}_{\infty}$ and that the sensing is performed continuously throughout the whole search. In real life, the solution to our problem can be used, e.g., as an efficient planner for a search and rescue scenario in which the mobile robot searches for victims after some catastrophic event.

Several papers [2, 3, 4, 5, 6, 7, 8, 9, 10] study similar versions of the problem. Although the initial assumptions (e.g., about the environment, sensor, time of sensing), formal definitions, and detailed solution methods differ amongst the authors, one general approach is shared — a two-stage decomposition of the problem:

1. the continuous problem is discretized by selecting a set of locations that completely cover the environment (cover-locations for short), and then
2. their visits' order is determined such that the expected time to find the *target* is minimized.

We follow the scheme as well, and for each stage, we either adopt from the literature or develop from scratch several solution methods, which are at the end evaluated on instances of the *mobile search*, and the best couple (combination) is proposed as the novel framework for the problem. We call the two stages the *discretization stage* (or just *the discretization*), and the *routing stage* (or just *the routing*), respectively.

In *the discretization*, we adopt methods based on *conforming constrained Delaunay triangulation* [11], convex partitioning of polygons [12], and *dual sampling* [13]; and develop a new hybrid method utilizing solutions of the *maximum area convex subset* (MACS) problem [14], *traveling salesman problem* (TSP) [15], and *touring polygons problem* (TPP) [16]. Also, we introduce a simple yet powerful filtering method able to improve sets of cover-locations generated by any of the previously mentioned methods.

The discrete problem solved as part of the *routing stage*, is defined on a complete graph by a set of pre-generated cover-locations and the shortest paths between them. Assuming sensing of the environment is performed exclusively on the cover-locations and that all locations have static (not changing as the search progresses) equal information gain, the problem is formulated as the *traveling deliveryman problem* (TDP) [17]. By the information gain of a particular location, we mean the amount of area newly covered (searched) when visiting that location. It is important to note that by formulating the *mobile search* this way, we admit severe inaccuracies since the assumptions of the TDP are very simplifying and, therefore, unrealistic. However, the TDP can be extended by many means to model the *mobile search* better. For example, the *graph search problem* (GSP) [18] considers differing gains of locations by introducing weights to the graph's nodes. Other extensions, e.g., to better model the robot's kinematics or consider sensing on the way between cover-locations, are introduced in this work. Also, replanning as the search progresses is proposed as a simple trick to deal with the non-static character of locations' gains. Besides the extensions, a significant portion of this thesis is dedicated to the original TDP, which the *operations research* community studied quite well compared to the extensions.

Suppose a set of customers in a city waiting for their deliveries, and travel times between each pair of them to be known. The TDP asks for a sequence of visits such that each customer is served exactly once, and the sum of all waiting times is minimized. This problem can be viewed as customer-oriented, as the one who provides the service seeks to satisfy the customers rather than minimizing own travel expenses [19]. A closely related and well-studied is the TSP, which aims in the just-mentioned opposite direction, i.e., it is so-called server-oriented. Both problems are known to be NP-hard for general metric spaces [17], and as their range of applications is multidisciplinary and wide, they have received much attention in the *operations research* literature in past decades (although we must say that the TSP significantly more than the TDP). For an exhaustive overview of the TSP and its applications, see [15]; practical applications of TDP that are traditionally mentioned by *operations research* authors are, e.g., customer-centered routing such as pizza-delivery or repairs of appliances [20], data retrieval in computer networks [21] or emergency logistics [22].

Quite recently, a different side of the scientific spectrum started to take notice of these problems and seek their efficient solutions; that is - robotics. One example for all is a *multi-goal path planning problem*, which is a robotic variant of the TSP with the edge costs as the length of the shortest paths connecting the locations of visits [23]. The efficient solution of this problem leads to an algorithm that helps a mobile robot to effectively build a map of an environment in which it operates or to patrol an area that is a-priori known to the robot. In fact, many variants and modifications of the TSP are often considered in robotics, e.g., *TSP with neighborhoods* [24], *generalized TSP* [25], or *orienteering problem* [26]. The TDP is no exception to this trend since it also has found its way to robotic problems. As mentioned previously, the TDP and its generalized version GSP can be used to formulate the *mobile search* as shown in [7, 8], for either known or unknown environment in which the robot operates.

The motivation behind thoroughly studying the TDP in this work is to solve it in the specific context of mobile robotics, characterized by the need to periodically replan a route while the robot moves and senses the environment. We aim to find a solution of the best possible quality while bounding the computational time by a given hard limit, so that the replanning can be done in a real time with a fixed frequency. These restrictions are different from those that authors of related works usually consider. The literature seems to follow two main streams in solving the TDP. Either the authors seek for an exact algorithm that will solve the problem to optimality [20, 27, 28], or their approach relies on heuristics and metaheuristics that are able to find good quality solutions (but not necessarily optimal) in *reasonable* computing time [29, 30, 31]. However, the term *reasonable* is often not well-specified. Usually, it merely holds that — the faster method, the better — as long as its average solution quality is comparable to the current state of the art. Nevertheless, this sort of a simple quality metric is not sufficient to tell which algorithm presented in the literature returns the best solution after t_{max} seconds.

Our aim w.r.t. the TDP is to systematically design a heuristic method and experimentally compare it to the current state-of-the-art method using various metrics. The emphasis is placed on results obtained under the hard time limit setting since this application scenario models the use in mobile robotics and is the most often left out by other authors. The considered time limits are in the range from 1 to 100 seconds. Nevertheless, some efforts are made to show that our method can compete with state of the art in the traditional sense. To have such a universally well-performing method, we use a general *run-time distribution* (RTD) [32] methodology to design it and tune its parameters. We consider several general improvement strategies during the design process, and within them, many combinations of their variable components. The proposed method is the best among all tested variants.

To summarize — in this work, we present a general solution to the *mobile search* problem, where a novel tailored approach to the TDP is integrated. Specifically, the key contributions are the following:

- We propose a novel way to discretize a 2D polygonal environment based on *computational geometry* and *combinatorial optimization* approaches. The discretization is shown to provide better results on *mobile search* instances than existing methods adapted from the literature.
- A new metaheuristic for the TDP, which takes into account limited computing time, is designed using a general RTD methodology, and *time-to-target* plots. The novel method is evaluated on several sets of standard benchmark instances used by the *operations research* community. It is shown to significantly outperform the current best approach from the literature under the hard limit settings with limits ranging from 1 to 100 seconds and still provide competitive results in the traditional sense and with cost targets corresponding to the best-known solutions worsened by about 1%.
- The metaheuristic is further extended to better model the *mobile search*, which is done by considering non-equal and non-static locations' information gains, an effort needed to turn the robot, and sensing on the way between the cover-locations.
- The extended metaheuristic and the novel discretization method is integrated into a software framework for the *mobile search*. Designing and implementing the framework is part of the work as well. We create a simple ideal simulator for the *mobile search*, which assumes that the robot's movements are composed of simple maneuvers: going ahead (in the current direction) with velocity v_{lin} and turning on the spot with velocity v_{ang}. The velocity constants are estimated based on simulations implemented in *Robot Operating System* (ROS) with existing commercial robot. Then, we generate two types of results from simulating the *mobile search*. For the first type, we use only our ideal simulator with tuned velocity constants; for the second type, we replace the

navigation part of our simulator with more realistic simulations performed with ROS. The proposed algorithms, i.e., the discretization method and the routing metaheuristic, are evaluated using these results on several instances of the *mobile search* and compared with alternative methods adapted from the literature.

This diploma thesis project is a free continuation of the subjects studied in our previous (bachelor's) thesis [33]. In the previous work, we develop a metaheuristic for the GSP that improves the results obtained by Kulich et al. [10]. Then we show that the metaheuristic can be used as part of a solution framework for the *mobile search*. This work, on the other hand, provides thorough study of the *mobile search* and broad spectrum of related problems. All solution approaches, methods, results, and even implementations and methodology presented in this work are either bright new or significantly improved or updated compared to those of the previous work.

The remainder of this chapter explains the *mobile search* problem in an accessible way without formal mathematical definitions, provides a background to some important chosen topics, and finally reviews related literature. The rest of this thesis is organized as follows. The main problems, the *mobile search* and the TDP, are defined in Chap. 2. Solution approach to both main problems and several related subproblems is described in Chap. 3. Properties of the proposed solution methods are evaluated and compared to the literature's approaches in Chap. 4. The last Chap. 5 is devoted to concluding remarks and reveals our publication plans with the work presented in this thesis.

1.2 Opening example

In this section, we yield an initial understanding and some insights into the main problem we study, the *mobile search*, in an accessible way without formal mathematical definitions. The formal definitions are established later in Sec. 2.1.

Let us illustrate the *mobile search* problem with an example in Fig. 1.1. Here, the environment is represented by the white space bounded by the square border and the C-shaped obstacle, both shown in black. The *target* is some tangible object that can be localized as a point somewhere in the environment, but its actual location is unknown and is therefore not shown in the picture. The robot is equipped with a sensor of 360° view and an unlimited visibility range, and its initial position marked s is at middle-bottom facing up. The objective is to plan and execute a path which covers the whole environment, i.e., every point of

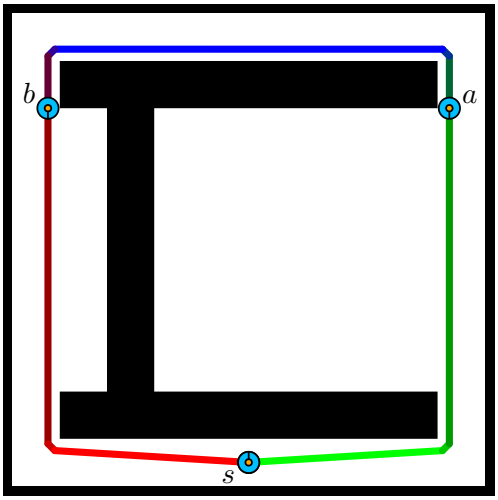
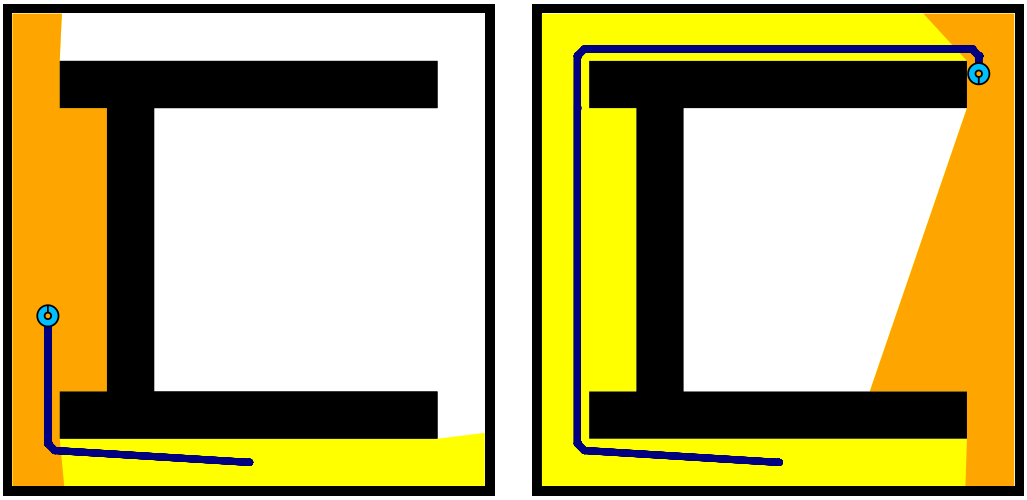


Figure 1.1: Opening example of the *mobile search*.

the environment would be seen (at least once) during the execution, to provide a 100 % guarantee the hidden object is found. However, different such paths may be differently efficient, as we show next. Back in the picture, there are two options displayed. The robot can either turn left, follow the red-blue path, and finish in position a, or it can turn right, track the green-blue path, and settle at position b. The two options and corresponding paths (routes, trajectories, &c) are further referred to as *the left* and *the right*, respectively. Now the question is — which option would be a better choice — *the left* or *the right*?

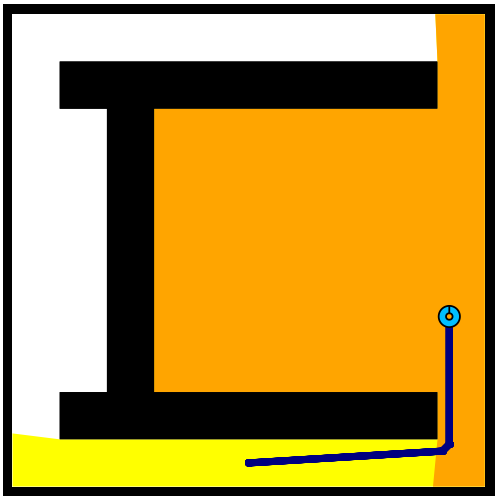
Before answering the question, let us give some perspective to the example. The environment is about 20×20 meters large, and the robot can move by a sequence of simple maneuvers consisting of going ahead (in the current direction) with velocity 1 m/s, and turning on the spot with velocity π rad/s. Furthermore, allow us to consider a general probabilistic case, where the object of interest is not hidden at some specific location but instead assumed to appear in a region of the environment with probability proportional to that region's area. That being said, to fully execute one or the other path (note the symmetry), the robot needs 46.5 seconds, and the probability that the object is found before a specific time is equal to the area of a covered region at that time divided by the area of the whole environment. Therefore, the probability is $\leq 100\,\%$ during the execution and is exactly 100 % at 46.5 s.

Now we are prepared to see Fig. 1.2, which shows covered portions of the environment at specific time ticks of the execution for both considered (*the left* and *the right*) paths. After 15 s, the robot searched only 32 % of the environment for *the left*, while for *the right* more impressive 71 %. A similar gap can be observed right before the end of the executions at 45 s — 62 % and 98 %, respectively. Based on these remarks, a thoughtful reader might have already rightly guessed that *the right* path is the right with respect to our problem. Also, the intuition works

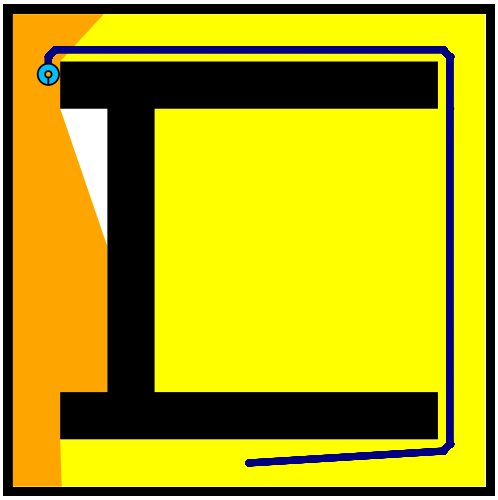


(a) : *Left* path, 15 s., 32 % covered.

(b) : *Left* path, 45 s., 62 % covered.



(c) : *Right* path, 15 s., 71 % covered.



(d) : *Right* path, 45 s., 98 % covered.

Figure 1.2: Execution of the search. A currently seen portions of the environment are shown in orange, and the covered regions are the union of orange and yellow. Already traversed parts of the planned paths are dark blue.

in this example. Since the big room (interior of the obstacle) can be seen as a whole by a single look (from certain positions) and takes nearly half of the entire environment, one may infer it would be a good idea to cover it first, then do the rest of the corridors. *The right* path follows precisely this notion, while *the left* seems to ignore it. *The left* path might be desirable if we assumed the object was hidden purposely somewhere behind a corner and certainly not in the big room's wide-open area. However, no such assumption is in place since all points of the environment are equally likely to exhibit the object. In other words, the probability of locating the object at specific coordinates is uniformly distributed over the environment's interior. An extension considering other than uniform probability distribution is possible but not considered in this work.

To conclude, *the right* path comprises a much more efficient search strategy since it covers large portions of space at the earliest.

1.3 Subject background

This section provides necessary backgrounds for some chosen topics addressed by this thesis and briefly explains a selection of related terms, theory, or methodology.

1.3.1 Art gallery problem

The *discretization stage* of the decoupled approach to the *mobile search* is related to the *art gallery problem* (AGP) [34], a well-known problem in *computational geometry*. It originates from a real-world task of guarding an art gallery with a minimal number of guards who together behold the whole gallery. Formally, the gallery is represented as a simple (has no holes and self-intersections) polygon P, and the guards by points in its interior. A set $G \subset P$ is said to guard P if, for every $p \in P$ there is some guard $g \in G$ such that the line segment between p and g is fully inside P. Chvátal published the first theorem [35] related to AGP in 1975, and since then, many other related theorems, algorithms, and variants of the problem were proposed by many authors. The environment's *discretization* that we consider can be viewed as a version of the AGP, where P corresponds to the environment and is allowed to have holes, and guards are the cover-locations and have limited visibility radius.

However, unlike the AGP, which seeks any minimal guards set, our problem's goal (optimization criterion) is hard to state. It is unclear which properties of a guard set would always yield good quality solutions to the *mobile search*. It is perhaps to ask whether such distinct properties can even exist since combined optimal solutions to the decoupled problems do not necessarily provide an optimal solution to the original problem. A better approach than decoupling, and in some sense the only correct, would be to solve the *mobile search* complexly as a whole. Some problems similar to the *mobile search*, e.g., the *watchman route problem* [36], can be tackled wholly, e.g., by utilizing *self-organizing maps* [37]. However, to the best of our knowledge, for the *mobile search* there are no such existing solutions in the literature, as well it would be too challenging to attempt the first by us at the moment. In later sections, we discuss this issue in more depth and suggest some partial solutions that compromise between optimality and tractability. We discuss more about the *watchman route problem* and its relation to the *mobile search* in Sec. 2.1.4.

1.3.2 Routing problems

The TSP, the TDP, and its variants mentioned in the introduction belong to a large general class of *combinatorial optimization* problems called *routing problems* [38], and within it, they are part of a group called *vehicle routing problems* (VRPs) [39]. VRPs ask for the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers. The city with customers is usually modeled as a graph $G = (V, E)$, where vertices V correspond to customers, depots, and other places in the city, and edges E correspond to road connections. VRPs differ from each other by the graph's characteristics (e.g., directed/undirected, complete/incomplete), considered optimization criteria (e.g., min. global transportation cost, min. customers' waiting times, min. number of vehicles, max. profit), and various constraints usually imposed by real-life applications (e.g., pick-up and delivery, time windows, limited vehicle capacities, number of available vehicles).

Within VRPs, the TSP and TDP share some common characteristics. Both are defined on a complete undirected graph (i.e., a fully inter-connected city with no one-way streets), assume only one available vehicle and no other constraints. A valid solution to both problems is a permutation of the graph's vertices. Their only difference is in the optimization criterion. The TSP minimizes the transportation cost and the TDP customers' waiting times while assuming both the cost and travel time are proportional to the traveled distance. The deliveryman can see the waiting times as a total latency of arrivals to customers, so the TDP is also called the *minimum latency problem* (MLP) [30]. However, the classical TDP is not the only problem that minimizes latency, as all its variants considered in this work do as well. Therefore, we group them all as *minimum latency problems* (MLPs) and use the term TDP only to describe one specific, the classical one.

1.3.3 Metaheuristics

"Metaheuristics, 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." [40]

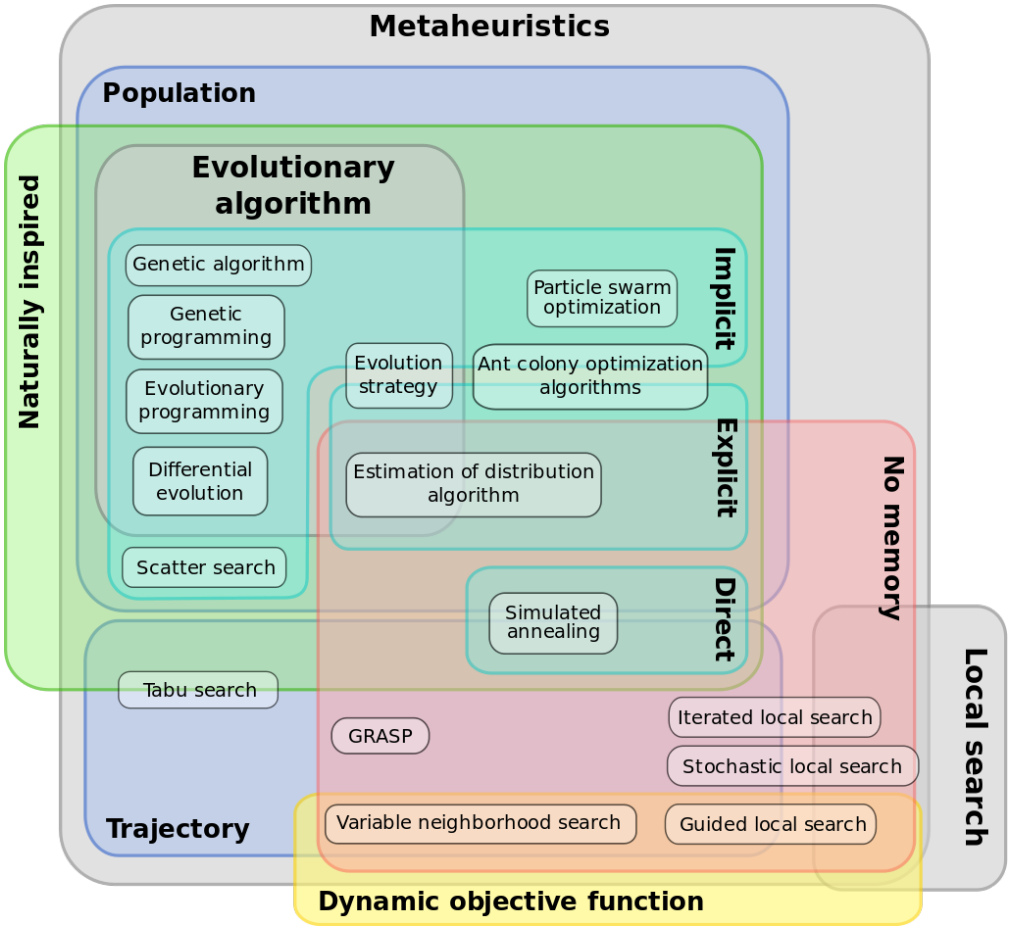


Figure 1.3: Metaheuristics and their classification; source: Wikipedia [41].

A graphical overview of metaheuristics that appear all across computer science and their classification according to several criteria is shown in Fig. 1.3. In this work, we consider several metaheuristics for the MLPs. They are all based on general schemes, which can be applied to a wide variety of problems. Here, we shortly describe the most relevant ones.

Variable neighborhood search (VNS) proposed originally by [42] is a single-start stochastic metaheuristic based on the idea of improving a single solution temporally even by some non-improving steps. In its scheme, two phases alternate: a shake which allows escaping local optimum and a local search phase, which descents towards one. Additionally, a systematic change of neighborhoods within the search is applied. *General VNS* (GVNS) is a variant that uses *variable neighborhood descent* (VND) in the local search phase. VND can be seen as a deterministic variant of VNS, which explores a solution space using several neighborhood structures, usually in a sequential order. *Greedy randomized adaptive search procedure* (GRASP), unlike VNS, is a multi-start process developed and established within the research community by many authors' works, e.g., [43, 44, 32].

Greedy randomized adaptive (GRA) construction heuristic is applied to each restart to create a new solution, which is then improved by VND, and the best overall solution is returned at the end. GVNS and GRASP have similarities and also significant differences. They both use VND as a local search method, and both are stochastic to be able to escape local optima - but in a different way. While GVNS randomly perturbates the best current solution (in the shaking phase), GRASP creates an entirely new one in a randomized fashion and starts the search from the beginning.

1.3.4 Run-time distribution

Hoos and Stützle [45] point out pitfalls related to stochastic methods evaluation and introduce a methodology for evaluating a certain class of algorithms called *Las Vegas algorithms*. However, the methodology can be extended to consider improving strategies such as TDP-solving metaheuristics as well. An algorithm $\mathcal{A}$ is said to be *Las Vegas algorithm* for problem class Π, if (i) whenever for a given problem instance $\pi \in \Pi$ it returns a solution, it is guaranteed to be valid, and (ii) on each given instance the run-time of $\mathcal{A}$ is a random variable. The authors classify three types of possible application scenarios for *Las Vegas algorithm* $\mathcal{A}$:

1. there are no time limits, i.e., we can afford to run the algorithm as long as it needs to find a valid (or of sufficient quality for some problems as TDP) solution;
2. there is a time limit t_{max}, which can be very small in case of real-time applications such as robotics;
3. the utility $U : \mathbb{R} \rightarrow [0, 1]$ of a solution depends on the time t needed to find it.

It is apparent that evaluating the performance of $\mathcal{A}$ in these scenarios must be done using different criteria for each. E.g., in the case of type 1, the mean time of several runs might suffice to characterize the run-time (rt) behavior roughly, but it is basically meaningless for type 2, which needs more adequate criteria such as $P(rt \leq t_{max})$ - the probability of finding a solution within the given time-limit. Additionally, we can observe that type 1 and 2 are special cases of the most general type 3, which can only be appropriately characterized by the run-time distribution function $rtd(t) = P(rt \leq t)$ or its approximation such as *time-to-target* (TTT) plots. The RTD was first used by Feo et al. [32] and further addressed by other authors [45, 46]. Hoos and Stützle [45] encourage to use the RTD to characterize the behavior of algorithm $\mathcal{A}$ completely and uniquely and stress out the possible pitfalls (such as imprecision or erroneous conclusions) when other simpler methodologies are used. In addition, from RTD, other criteria,

like the mean run-time, its standard deviation, median, percentiles, or success probabilities $P(rt \leq t_i)$ for arbitrary time-limits t_i, can be extracted.

As we mentioned earlier, Hoos and Stützle originally proposed the methodology for *Las Vegas algorithms*, which for a given instance, either return a valid solution in a finite time rt or do not find any (then $rt = \infty$). However, in the case of TDP, the best metaheuristics are almost always improving strategies, meaning they construct a valid solution in the early stage of their run-time and spend the rest of the time improving it (without the loss of validity). In order to apply the RTD to improving strategies a solution cost goal c_{goal} needs to be considered. The function $rtd(t) = P(rt \leq t)$ is then seen as the probability that the algorithm finds a solution with a cost at least as good as c_{goal} in time $rt \leq t$. In other words, if only a solution $\mathcal{X}$ with the cost $cost(\mathcal{X}) \leq c_{goal}$ is considered valid, then the TDP-solving algorithm is a *Las Vegas algorithm* for the TDP. The use of this technique is recommended in [47] for many problems (including TSP) and the value of c_{goal} is often chosen to be 1% worse than the currently best known solution by the authors.

To wrap up, the single most useful (thanks to its universality) way to characterize the run-time of stochastic solution methods for the TDP, which is relevant even in our context of robotics (scenario type 2), seems to exist. The problem is that it is barely used in the related literature. What is used instead are the best and mean solution costs and average values of CPU times, which are higher by far (especially for medium and large instances) from the time limits imposed by the robotic context (order of units of seconds). Since neither $rtd(t)$ nor $P(rt \leq t_{max})$ is used to characterize someone's algorithm, we can conclude that their results can be relevant only to the solution scenario of type 1, i.e., limit-less computation times, while for the other two, no valid conclusions can be made. The call for heuristic methods that solve the TDP and yet perform well in all application scenarios 1-3 is in place. It comes from the experience that the computational time to solve the problem is often limited, or the usefulness of a solution is dependent on the time needed to obtain it.

1.3.5 Time-to-target plots

Consider an instance $\pi \in \Pi_{\mathcal{O}}$ of an optimization problem class $\Pi_{\mathcal{O}}$, a set of all its valid solutions $\mathcal{H}(\pi)$, a cost function $cost : \mathcal{H}(\pi) \mapsto \mathbb{R}_0^+$, an optimal solution cost $c^\star = \min_{\mathcal{X} \in \mathcal{H}(\pi)} cost(\mathcal{X})$, and a target[1] cost value $c_{goal} \in \mathbb{R}_0^+ : c_{goal} \geq c^\star$. We call algorithm $\mathcal{A}_{\mathcal{I}}$ for $\Pi_{\mathcal{O}}$ a *Las Vegas improving algorithm*, if (i) whenever for a given π it returns a solution $\mathcal{X}$, it is guaranteed to be valid with cost $cost(\mathcal{X}) \leq$

[1] Not to be confused with the *target* (physical object of interest) as in *target detection* and related problems.

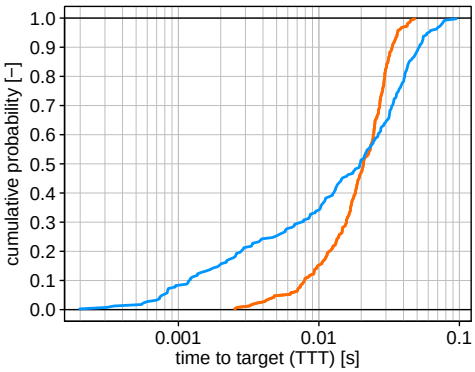


Figure 1.4: An illustration example of TTT-plots for two algorithms.

c_{goal}, and (ii) for each $\pi \in \Pi_{\mathcal{O}}$ the run-time rt of $\mathcal{A}_{\mathcal{I}}$ is a random variable. The algorithm $\mathcal{A}_{\mathcal{I}}$ is run n_{run} times on the fixed instance π. The runs are assumed to be independent, i.e., the random number generator is initialized with a different seed every time. The rts of all runs are recorded and saved and then used to produce a TTT-plot. TTT-plots construction is well-described in [47] and we follow the same methodology shown by the authors. After finishing the last run, the recorded rts are sorted in increasing order and the probability $p_i = (i - 1/2)/n_{run}$ is associated with each i-th sorted rt_i, for $i = 1, \ldots, n_{run}$. The meaning of p_i can be understood as follows: p_i is the probability that the algorithm finds a solution at least as good as the target c_{goal} in at most rt_i seconds. Finally, a TTT-plot is constructed by plotting all points (rt_i, p_i). Clearly, the TTT-plot is an approximation of the cumulative RTD capable of characterizing the run-time behavior of *Las Vegas improving algorithms* as we defined them. Additionally, for practical reasons, we further consider an upper run-time limit t_{max} for the given instance. By convention, we assume that whenever the method's running time reaches t_{max}, it stops, and $rt = \infty$ is recorded. The convention assures, that an experiment of n_{run} independent runs will finish in reasonable time, more specifically, in $n_{run}t_{max}$ seconds in the worst case. Clearly, the upper bound t_{max} must be sufficiently large to obtain valid results.

One obvious drawback of TTT-plots is that whenever two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ are evaluated on the same instance and their TTT-plots are superimposed, it might not be clear on the first sight, which algorithm performs better and by how much. To compare the algorithms productively, some adequate metric must be introduced. Let RT_1 and RT_2 be random variables representing the time needed by algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively to find a solution as good as the given target value. Let $p_{12} = P(RT_1 \leq RT_2)$ be the probability, that the random variable RT_1 takes a value smaller or equal to RT_2. Assuming that both algorithms stop when (and only if) they find a solution at least as good as the target, we can say that $\mathcal{A}_1$ performs better than $\mathcal{A}_2$ if $p_{12} > 0.5$. An iterative procedure to compute p_{12} with arbitrary small approximation error for two algorithms following general cumulative probability distributions is introduced in [48]. Later, Ribeiro and Rosetti [49] develop a program to compute the approximation of p_{12} from provided

TTT-plots of the two algorithms. In this work, we use a computation inspired by their program. For the theory and related issues, see [48, 50, 49] or the overview in [47].

For an illustration example of TTT-plots for two algorithms: *the blue* and *the orange*, see Fig. 1.4. The two algorithms solve a particular 50-customer instance of the TDP given the same target cost, in this case, the optimum. The estimated probability that they return the optimal solution before time 20 ms is about 50 % for both. Note that *the blue* is superior for time limits TTT < 20 ms, but for less strict time limits, the probability of returning the optimum in time is higher for the *the orange*. The probability that *the blue* algorithm will return solution of required quality before the *the orange* is $P(RT_{blue} \leq RT_{orange}) = 51.9\,\%$.

1.4 Related literature review

The ***mobile search*** problem in a 2D polygonal environment is solved in a two-stage process by Sarmiento et al. [2, 3]. Their formulation assumes an unlimited visibility range for the robot's sensor, a discrete sensing only at selected locations, and a known probability density function characterizing the searched object's unknown location. They first transform the continuous problem into a discrete case by selecting a set of exclusive sensing locations that completely cover the environment, then determine the order of their visits. The order is determined by a greedy algorithm that gradually adds a location with a maximal utility value. The location utility is a ratio of a gain of visiting the location and an effort needed to visit it. Furthermore, a breadth-first search with heuristic pruning was introduced as an extension of the greedy algorithm. The extension iteratively constructs all possible defined length routes, fixes the most promising one, and starts the next search from this route as a prefix. In [4], the same authors study a multi-robot variant of the problem, and in [5], they extend it by considering continuous sensing. For the last variant, they again propose a two-step approach — first, a set of critical curves is identified and then considered in a simplified search-path problem where the curves are refined and locally optimized.

Another decomposition of the problem, similar to the previous one, is introduced in Lv et al. [6]. The environment is split into subregions for which a center position and its importance evaluation function are established. The center positions are then visited in an order determined by an improved particle swarm algorithm.

Kulich et al. consider search in an unknown environment in [7], a problem similar to the exploration task. However, they show that search and exploration objectives are dissimilar and propose a combination of a frontier-based approach and a modified depth-first search algorithm with pruning and limited branching to determine the order in which the frontiers are investigated. Later they extend the work by formulating the search problem as the GSP and introduce a more sophisticated algorithm based on the GRASP metaheuristic in [8]. Shortly after, they also discuss a multi-robot version of the problem [9]. In their most recent work [10], they develop for the multi-version of the GSP a metaheuristic based on a combination of GRASP and VND.

In the *routing* stage of solving the *mobile search* we deal with MLPs and mostly with the **TDP**. Two noticeable leading courses are apparent in solving the TDP in the *operations research* community. The first one aims to find the optimal solution to the problem; however, it is limited to small instance sizes due to infeasible computing times. The second major course, a more relaxed one, seeks a solution that is only close enough to optimum in exchange for much lower computational times. Heuristics, often based on some more general search strategies (metaheuristics), are the core solution methods in this area. Approximation algorithms, which lie somewhere in the middle, are also known for the TDP. These methods give approximate solutions, but, unlike heuristics, with a theoretically proven guarantee of performance.

Early **exact algorithms** proposed by [51, 52] rely on non-linear integer formulations in which a Lagrangian relaxation is used to derive lower bounds. Fischetti et al. [20] develop *integer linear programming* (ILP) formulation and new theoretical results on the matroidal structure of a class of combinatorial problems. The results are used to derive lower bounds for the TDP and are embedded into an enumerative algorithm capable of solving 60-vertices instances to optimality. Some other ILP formulations, as well as *mixed integer linear programming* (MILP) formulations, and exact algorithms are proposed in [53, 54, 55, 56, 28]. In the last mentioned, Naeni and Salehipour develop a MILP model that benefits from position-based variables. On the set of 70 randomly generated instances of sizes 10-50, they show their model can deliver the largest number of the best solutions in a shorter time compared to most of the already mentioned models [20, 53, 54, 55].

In addition to TDP-specialized solutions, several ILP formulations and exact algorithms are developed for the *time-dependent traveling salesman problem* (TDTSP), a generalization of both the TSP and the TDP. Some of these formulations are proposed in [57, 58, 59, 27, 60, 61, 62]. Overall, the strongest algorithm is the branch-cut-&-price developed by Abeledo et al. [59, 27], capable of solving almost all instances from the TSPLIB [63] with up to 107 vertices within the limit of 48 hours.

Approximation algorithms for the TDP on a tree or on a general metric graph are developed in [64, 65, 66, 21, 19, 67, 68, 69, 70, 71, 72]. The researchers who develop approximation algorithms focus mostly on lowering the approximation factor or computational complexity of their algorithms; however, computational results on benchmark instances are usually not present in their works. The lowest approximation factors in the literature are 3 [72] and 3.59 [67] for the tree and the general case respectively.

The heuristic approach mostly relies on more general search strategies — **metaheuristics** — especially VNS and GRASP. Salehipour et al. [29] propose a GRASP that embeds either VND or VNS and evaluate both variants on a set of randomly generated benchmark instances of sizes in a range from 10 to 1000. Silva et al. [30] later present a simple and effective metaheuristic called GILS-RVND, which is based on the combination of GRASP and *iterated local search* (ILS). It improves all the results obtained by [29] on their instances and finds new best solutions for two of TSPLIB [63] instances. Mladenović et al. [31] propose a GVNS, able to improve the previous results obtained by [29] as well; however, GILS-RVND still performs slightly better in terms of solution quality. Ban et al. [56] suggest a metaheuristic algorithm combining *tabu search* (TS) and VNS and show that it compares well with the state-of-the-art algorithms [29, 30] in the quality of obtained solutions. The TS-VNS, however, does not improve the bar set by the GILS-RVND in the matter of computional time.

To the best of our knowledge, since its publication in 2012 to this day, the GILS-RVND by Silva et al. has been the one heuristic method providing the best trade-off between its simplicity, solution quality, and computational time in the literature. Thanks to its affable characteristics, it more recently has been chosen by other authors as the base method for their improvement ideas. Rios [73] propose versions of GILS-RVND for parallel computing in CPU/GPU hybrid systems. Santana et al. [74] improve GILS-RVND by means of *data mining* (DM) techniques. Their new hybrid method, called *multi-DM GILS-RVND* (MDM-GILS-RVND), utilizes the *frequent itemset mining* (FIM) technique to gather segments of high-quality solutions in the first half of GILS-RVND iterations. In the other half, the segments are used to construct new initial solutions with every other restart. MDM-GILS-RVND is shown to perform almost equally as GILS-RVND on small instances ($n \leq 50$), better in terms of computational time on medium instances ($50 < n \leq 200$), and better in both terms of time and solution quality on large instances ($200 < n$).

Chapter 2

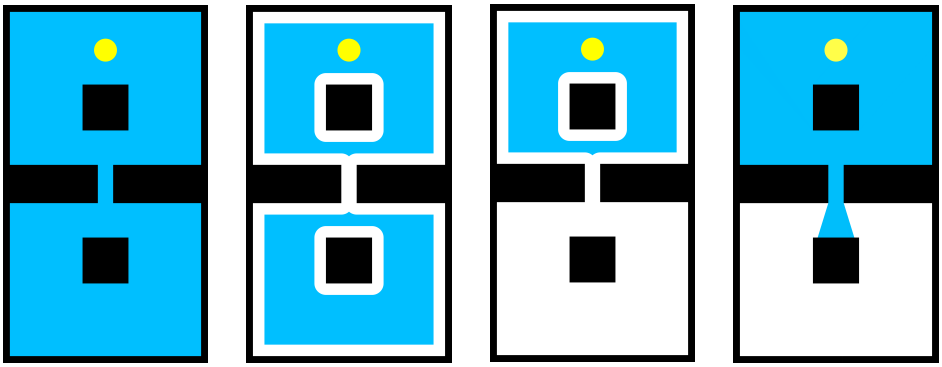
Problems' definitions

2.1 Mobile search

This section provides a general mathematical definition of the *mobile search* problem based on Sarmiento et al. [5] and a more practical formulation based on Kulich et al. [8]. First, we define some auxiliary structures in Subsec. 2.1.1, then the *mobile search* formulations follow in Subsec. 2.1.2 and 2.1.3, respectively, and finally in Subsec. 2.1.4, we discuss the difference between the *mobile search* and a similar problem which minimizes the time of search in the worst case.

2.1.1 Auxiliary definitions

We assume a robot that operates in an environment $\mathcal{W} \subset \mathbb{R}^2$ that is static, enclosed, and 2D polygonal, i.e., modeled as an over time non-changing single polygon with holes (obstacles). The obstacles and an outer border of the environment, denoted as $\mathcal{O} = \mathbb{R}^2 \setminus \mathcal{W}$, generate motion and visibility constraints for the robot, i.e., it can neither go nor see outside of the environment or over the obstacles. The robot's configuration $q = (x, y, \phi)$ is expressed by 2D coordinates $(x, y) \in \mathbb{R}^2$ of its footprint's center, and its heading $\phi \in [0, 2\pi)$. Let $\mathcal{C} = \mathbb{R}^2 \times \mathsf{SO}(1)$ be the *configuration space* (set of all possible configurations), and $\mathcal{A}(q) \subset \mathbb{R}^2$ set of all points representing the robot determined by given configuration $q \in \mathcal{C}$. Then, the set of collision-free configurations is defined as $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$ where $\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$. We also denote the set of all environment's points visible by



(a) : $\mathcal{W}$ **(b)** : $\mathcal{C}_{free}$ **(c)** : $\mathcal{C}^0_{free}$ **(d)** : $\mathcal{W}^0_{vis}$

Figure 2.1: Sets $\mathcal{W}$, $\mathcal{C}_{free}$, $\mathcal{C}^0_{free}$, and $\mathcal{W}^0_{vis}$ for a circular robot with omnidirectional sensor and unlimited visibility range (i.e., the same setup as in the opening example 1.2) displayed in blue color. Only the x and y coordinates are shown for $\mathcal{C}_{free}$ and $\mathcal{C}^0_{free}$. Obstacles $\mathcal{O}$ and the robot's initial footprint $\mathcal{A}(q_0)$ are displayed in all figures in black and yellow, respectively.

the robot from given configuration $q \in \mathcal{C}$ as $\mathcal{V}(q) \subset \mathcal{W}$. For all $q = (x, y, \phi) \in \mathcal{C}$ we naturally assume that $(x, y) \notin \mathcal{W}$ implies $\mathcal{V}(q) = \emptyset$. Let us further assume some initial configuration $q_0 \in \mathcal{C}_{free}$ of the robot, and let $\mathcal{C}^0_{free}$ be the *connected component* of $\mathcal{C}_{free}$ that contains q_0. Then, given initial configuration q_0, we define the *possibly-visible subset* of environment $\mathcal{W}$ as

$$\mathcal{W}^0_{vis} = \bigcup_{q \in \mathcal{C}^0_{free}} \mathcal{V}(q). \tag{2.1}$$

For a visual example of $\mathcal{W}$, $\mathcal{C}_{free}$, $\mathcal{C}^0_{free}$, and $\mathcal{W}^0_{vis}$ see Fig. 2.1.

In our formulation of the *mobile search*, we assume the environment $\mathcal{W}$ and robot's initial configuration q_0 are known, but we have zero information about the searched object's location. Regarding the object, although its location $p_? \in \mathcal{W}$ is a-priory unknown, we must naturally assume

$$p_? \in \mathcal{W}^0_{vis}; \tag{2.2}$$

otherwise, the object might never get into the robot's field of view. Note that this assumption is not necessary for environments where $\mathcal{W} = \mathcal{W}^0_{vis}$ (see Fig. 1.1 for an example of such environment). In relation to the assumption, we call $\mathcal{W}^0_{vis}$ the *admissible region for search* or just the *admissible region*.

Furthermore, we define a trajectory τ as a continuous curve of finite length in the configuration space that is parameterized by time, i.e., $\tau : [0, t_{stop}] \mapsto \mathcal{C}$, $t_{stop} \in \mathbb{R}^+$. Assume a specific time $t \leq t_{stop}$ and let

$$\mathcal{W}_{cov}(\tau, t) = \bigcup_{t' \in [0,\, t]} \mathcal{V}(\tau(t')) \tag{2.3}$$

be the subset of the environment that has been seen by the robot as it moves along τ until time t. We say that at time t the trajectory τ has covered the region $\mathcal{W}_{cov}(\tau, t)$ and, in general, we use the term *cover* to denote that the robot has sensed (seen) a certain portion of the environment. We call a trajectory $\tau : [0, t_{stop}] \mapsto \mathcal{C}$, $t_{stop} \in \mathbb{R}^+$

- *collision-free* iff $Range(\tau) = \mathcal{C}_{free}$, i.e., $\tau : [0, t_{stop}] \mapsto \mathcal{C}_{free}$,
- *starting at* q_0 iff $\tau(0) = q_0$, and
- *maximally covering* iff it covers the whole *admissible region* before time t_{stop}, i.e., $\mathcal{W}_{cov}(\tau, t_{stop}) = \mathcal{W}^0_{vis}$.

2.1.2 General formulation

Consider a known environment $\mathcal{W}$ and inside a robot at initial configuration q_0 that at time $t = 0$ starts executing a *collision-free* trajectory $\tau : [0, t_{stop}] \mapsto \mathcal{C}_{free}$ *starting at* q_0 while searching for a *target* whose location $p_? \in \mathcal{W}^0_{vis}$ is unknown. The *target* is some tangible static object of interest that is localized (associated with concrete coordinates) at the moment it is found which is when it first appears in the robot's field of view. Let the time at which the robot first sees the object of interest be described by a random variable T. The probability that the robot finds the object before any time t is given by the *cumulative distribution function* (CDF) of the random variable T, i.e.,

$$F_T(t) = \int_0^t f_T(t')\,\mathrm{d}t' = P(T \leq t). \tag{2.4}$$

Since the probability clearly depends on the trajectory followed by the robot, we define the CDF along a given trajectory τ as

$$F_{T\,|\,\tau}(t\,|\,\tau) = \int_{\mathcal{W}_{cov}(\tau,t)} f_{XY}(x, y)\,\mathrm{d}x\,\mathrm{d}y, \tag{2.5}$$

where $f_{XY}(x, y)$ is the *probability density function* (PDF) for the object's location. Then based on the general relation between CDF and PDF of random variable X — $F_X(t) = \int_{-\infty}^{t} f_X(t')\,\mathrm{d}t'$ — we can retrieve the $f_{T\,|\,\tau}$ from $F_{T\,|\,\tau}$. Under the assumption that τ is *maximally covering*, the expected (mean) [75] time to find the object can be computed as

$$E[T\,|\,\tau] = \int_0^{t_{stop}} t \cdot f_{T\,|\,\tau}(t\,|\,\tau)\,\mathrm{d}t. \tag{2.6}$$

The objective of the *mobile search* is finding the *maximally covering collision-free* trajectory $\tau^\star$ *starting at* q_0 that minimizes the expected time to find the object, i.e.,

$$\tau^\star = \arg\min_{\tau}(E[T \,|\, \tau]). \tag{2.7}$$

The *mobile search* is an infinite-dimensional optimization problem whose discretized version is NP-hard, as shown in [2].

2.1.3 Practical formulation

Note that the previous formulation of the *mobile search* is general enough to consider arbitrary PDF for the object's location and a robot of any shape and size equipped with a sensor with an arbitrarily shaped field of view. However, to tackle the problem in any feasible way, we must further concretize it by several more assumptions.

In some applications, there are reasons to believe that some object's locations are more likely than others. For example, if we are searching for keys at our home, we probably start looking at places where we usually leave them and discard whole rooms such as the bathroom. However, in our work, we do not consider these applications and instead we assume that all points of the *admissible region* are equally likely to exhibit the object. This assumption known in the literature as the *principle of insufficient reason* [76, 77] implies modeling the object's location by a uniform PDF. This yields a specific formula for computing the probability of finding the object prior time t while following trajectory τ:

$$P(T \leq t \,|\, \tau) = F_{T\,|\,\tau}(t \,|\, \tau) = \frac{a(t \,|\, \tau)}{a_{total}} = \frac{Area(\mathcal{W}_{cov}(\tau,\, t))}{Area(\mathcal{W}^0_{vis})}, \tag{2.8}$$

where $a(t \,|\, \tau) = Area(\mathcal{W}_{cov}(\tau,\, t))$ is the area of a region covered before time t, and $a_{total} = Area(\mathcal{W}^0_{vis})$ is the area of the whole *admissible region for search*.

In practical robotics, sensing and planning are performed in discrete times $\boldsymbol{t} = (t_1 = 0, t_2, \ldots, t_{stop})$; therefore, we can rewrite the equation (2.6) as

$$E[T \,|\, \tau] = \sum_{k=1}^{|t|} t_k \cdot p(t_k \,|\, \tau), \tag{2.9}$$

where $p(t_k \,|\, \tau)$ is the probability of finding the object of interest at specific time t_k. Here we must revisit our previous statement (from the introduction) that our variant assumes that the sensing is performed continuously throughout the whole search. The statement is true in a manner that sensing is not confined to a narrow

set of restricted sensing locations (such as in [3]) but is performed during the whole execution of the search independently on the robot's position. However, the continuity here is in question since we discretize the sensing in time according to Eq. 2.9. Therefore, we adjust the statement by declaring that continuous sensing is approximated by sensing with a high frequency, which is reasonable since continuous sensing cannot be realized in real-world scenarios anyway.

Applying assumption (2.8) to compute $p(t_k \mid \tau)$ leads to the following formula:

$$p(t_k \mid \tau) = \frac{a_{new}(t_k \mid \tau)}{a_{total}} = \frac{Area(\mathcal{W}_{new}(\tau, t_k))}{Area(\mathcal{W}^0_{vis})}, \tag{2.10}$$

where $\mathcal{W}_{new}(\tau, t_k) = \mathcal{W}_{cov}(\tau, t_k) \setminus \mathcal{W}_{cov}(\tau, t_{k-1}) = \mathcal{V}(\tau(t_k)) \setminus \mathcal{W}_{cov}(\tau, t_{k-1})$ is the newly covered region at time t_k and $\mathcal{W}_{cov}(\tau, t_0) = \emptyset$.

We further specify the problem by raising assumptions about the robot and its sensor. The robot's footprint $\mathcal{A}(0, 0, 0)$ is a circle with radius $r_\mathcal{A} \in \mathbb{R}^+_0$ (reduced to a single point for $r_\mathcal{A} = 0$) centered at the origin. Regarding the robot's movement, traversing any straight line segment fully inside $\mathcal{C}^0_{free}$ and rotating on the spot are possible (other movements are allowed, but not necessary), which means that any configuration $q_i \in \mathcal{C}^0_{free}$ is reachable from any other configuration $q_j \in \mathcal{C}^0_{free}$. Combining the two last assumptions (about the robot's shape and movement) implies that $\mathcal{C}_{free}$ can be computed by offsetting [78] the environment $\mathcal{W}$ by radius $-r_\mathcal{A}$ to get the x, y coordinates and considering the whole interval $[0, 2\pi)$ for the possible headings ϕ.

Regarding the field of view, the robot is equipped with an omnidirectional sensor with a limited visibility range $r_\mathcal{V} \in \mathbb{R}^+_\infty$ (this includes the case where the range is, in fact, unlimited for $r_\mathcal{V} = \infty$), placed in its center, i.e., the sensor's coordinates are identical to the robot's. This yields a recipe for computing the currently *sensed region* $\mathcal{V}(q)$ of the environment while the robot is at position (configuration) $q = (x, y, \phi)$:

$$\mathcal{V}(q) = \mathcal{C}_\circ(x, y, r_\mathcal{V}) \cap \mathcal{P}_{vis}(x, y). \tag{2.11}$$

Here $\mathcal{C}_\circ(x, y, r_\mathcal{V})$ is a circle with radius $r_\mathcal{V}$ centered at (x, y), and $\mathcal{P}_{vis}(x, y)$ is a *visibility polygon* [79] computed inside $\mathcal{W}$ with (x, y) as a seed. *Visibility polygon* for a given seed is a polygon whose all points are *visible* from the seed. One point p_i is *visible* from another point p_j iff the segment which connects them is fully inside $\mathcal{W}$. For examples of *visibility polygons* see the orange regions in Fig. 1.2. Let us further note that since our algorithms introduced later work with objects represented as polygons, i.e., sequences of points, we approximate the circle $\mathcal{C}_\circ(x, y, r_\mathcal{V})$ by a regular polygon with n_{poly} edges that could fit inside the circle. Most times, we use $n_{poly} = 16$.

Finally, for simplification purposes and to neglect possible confusion from $\mathcal{W}$, $\mathcal{C}_{free}$, $\mathcal{C}^0_{free}$, $\mathcal{W}^0_{vis}$ structures, we consider only such environments and robot's radii $r_{\mathcal{A}}$ that together produce $\mathcal{C}_{free}$ with only a single *connected component*; therefore, $\mathcal{C}^0_{free} = \mathcal{C}_{free}$. Similarly, we consider appropriate pairs of environments and visibility radii $r_{\mathcal{V}}$, such that the coverage of the whole environment is possible, i.e., $\mathcal{W}^0_{vis} = \mathcal{W}$.

2.1.4 Expected vs. the worst time

Let us note the difference between minimizing the expected time $E[T \mid \tau]$ as in the *mobile search* and minimizing the time it would take to find the object in the worst case. The latter's objective is to find the shortest (by means of time; or length if they are proportional) *maximally covering* (*collision-free*, &c) trajectory, which is a problem known in the literature as the *watchman route problem* (WRP). The WRP does not prioritize any parts of the environment, and the rate at which new areas are covered is unimportant. On the other hand, the *mobile search* tries to gain the probability mass of finding the object as quickly as possible. For the uniform PDF, the best strategy is to cover large portions of the environment at the earliest, as shown in the opening example (Sec. 1.2). For a given environment, the optimal strategy w.r.t. the WRP is usually not the best for the *mobile search*. For an example of this situation, return to the opening example and Fig. 1.2. There, both strategies, i.e., *the left* and *the right*, are equally good w.r.t. the WRP, as they are both the same length (seen from the symmetry), but as we argue in Sec. 1.2, *the right* strategy, compared to *the left*, is significantly better for the *mobile search*, as it covers the large room early in the execution.

2.2 Traveling deliveryman problem

In the *routing* stage of solving the *mobile search*, we deal with MLPs and mostly with the TDP as noted in previous sections. The TDP is a *combinatorial optimization* problem whose definition is much more straightforward than the *mobile search* definition.

The *traveling deliveryman problem* (TDP), also known as the *repairman problem*, or the *minimum latency problem*, is formally described by:

- $G = (V, E)$: a complete undirected graph with N vertices $V = \{v_1, \ldots, v_N\}$ in which every pair of distinct ones $v_i \neq v_j$ is connected by a unique edge $e_{i,j} = (v_i, v_j) \in E$.
- $d : E \to \mathbb{R}_0^+$: a non-negative cost $d(i, j)$ associated with each edge $e_{i,j} \in E$ representing a length of the shortest path (or travel time) from v_i to v_j. The costs are assumed symmetrical, i.e., $d(i, j) = d(j, i)$.
- $s \in V$: a starting vertex (depot) of the deliveryman; all other vertices represent the customers.

Let the sequence of vertices $\mathcal{X} = (x_0 = s, x_1, \ldots, x_n)$, where $n = N - 1$ is the number of customers, be a *Hamiltonian path* in G starting from the depot. *Hamiltonian path* is a path in a graph, i.e., sequence of vertices of that graph, that visits each vertex exactly once. Furthermore, let $d(x_i, x_j)$ be the cost of an edge between i-th and j-th vertex in $\mathcal{X}$. The cumulative cost to reach k-th vertex in the sequence $\mathcal{X}$ is defined as

$$\delta_k^{\mathcal{X}} = \sum_{i=1}^{k} d(x_{i-1}, x_i) \ . \tag{2.12}$$

Finally, the total cost of $\mathcal{X}$ is defined as

$$cost(\mathcal{X}) = \sum_{k=1}^{n} \delta_k^{\mathcal{X}} = \sum_{k=1}^{n} \sum_{i=1}^{k} d(x_{i-1}, x_i) \ . \tag{2.13}$$

The objective of the TDP is to find an optimal path $\mathcal{X}^*$ that minimizes the cost, i.e.,

$$\mathcal{X}^\star = \underset{\mathcal{X} \in \mathcal{H}(\pi)}{\arg\min}\, cost(\mathcal{X}), \tag{2.14}$$

where $\pi = (G, d, s)$ is an instance of the TDP and $\mathcal{H}(\pi)$ is the set of all *Hamiltonian paths* in graph G starting in s. Note that we assume an open variant of the problem,

i.e., the travel from the last customer back to the depot, is not considered. However, a *Hamiltonian cycle* instead of the *path* is often deemed in the literature. In that case, the total cost (2.13) would be defined as $cost(\mathcal{X}) = \left(\sum_{k=1}^{n} \delta_k^{\mathcal{X}}\right) + \delta_n^{\mathcal{X}} + d(x_n, x_0)$.

Regarding the *mobile search*, the cover-locations correspond to TDP customers, and the edges of the TDP graph represent the shortest paths between the locations. Lengths of the shortest paths define the costs d, and the initial position of the robot is represented by the depot s.

Chapter 3

Solution approach

3.1 General approach to the search

A general solution to the *mobile search* problem is described in this section. The sections coming after are dedicated to specific components of the solution or their improvements. We use the same notion and symbols as in Chap. 2 wherever possible. We kindly encourage the reader to review the chapter for some of the definitions.

The most natural way of applying the *mobile search* in real-life scenarios such as *search and rescue* would be the following. The robot is placed in an environment, supplied with its complete map, and given the *target's* description. The mission is to sweep the whole environment, find the *target* and report its location, or report that the *target* could not be found. The robot must efficiently plan the search, recognize the *target* once it appears in the sensor's field of view, and have many other smart or lower-level functions to ensure mission success. Those include, but are not limited to, localization, navigation, and motion control. Since our focus is the efficient planning, we consider everything else as given.

From the planner's perspective, we describe the *mobile search* mission algorithmically in Alg. 3.1. Here, the inputs are the environment, the robot's initial configuration, and some of its features. The output is the searched object's location and the total time of the mission. Configuration space structures $\mathcal{C}_{free}$, $\mathcal{C}^0_{free}$, and the *admissible region for search* $\mathcal{W}^0_{vis}$ are computed on line 1 of the algorithm. We consider such combinations of environments and robot's features that $\mathcal{C}_{free} = \mathcal{C}^0_{free}$,

Algorithm 3.1: *Mobile search*: mission

Input: $\mathcal{W}$ ... environment, q_0 ... robot's initial configuration,
$r_{\mathcal{A}}$ ... robot's radius, $r_{\mathcal{V}}$ ... sensor's range, f ... sensing frequency
Output: l_{object} ... location of the searched object,
t_{search} ... total time of the search

1 Compute $\mathcal{C}_{free}$, $\mathcal{C}^0_{free}$, $\mathcal{W}^0_{vis}$ structures.
2 $\boldsymbol{l}^{set} \leftarrow$ `Discretize(`$\mathcal{W}^0_{vis}$, $r_{\mathcal{V}}$, $\mathcal{C}^0_{free}$`)`.
3 $\boldsymbol{l}^{seq} \leftarrow$ `Plan(`$\boldsymbol{l}^{set}$, $\mathcal{W}^0_{vis}$, $r_{\mathcal{V}}$`)`.
4 $l_{object} \leftarrow$ (`NaN`, `NaN`), $t_{search} \leftarrow 0$, $\mathcal{W}_{cov} \leftarrow \emptyset$ ▷ `initialize`
5 **while** $\boldsymbol{l}^{seq}$ *is not empty* **do**
6 Pop the first location l_{goal} from $\boldsymbol{l}^{seq}$.
7 Start navigating the robot towards l_{goal} by the shortest path.
8 **repeat**
9 Get the currently *sensed region* $\mathcal{V}$ from sensor readings.
10 **if** *the searched object appears in* $\mathcal{V}$ **then**
11 Localize the object as l_{object}.
12 **return** l_{object}, t_{search} ▷ `report success`
13 $\mathcal{W}_{cov} \leftarrow \mathcal{W}_{cov} \cup \mathcal{V}$
14 $t_{search} \leftarrow t_{search} + f^{-1}$
15 **until** l_{goal} *is reached*
16 $\boldsymbol{l}^{seq} \leftarrow$ `RePlan(`$\boldsymbol{l}^{seq}$, $\mathcal{W}^0_{vis}$, $r_{\mathcal{V}}$, $\mathcal{W}_{cov}$`)` ▷ `optional`
17 **return** l_{object}, t_{search} ▷ `report failure`

$\mathcal{W} = \mathcal{W}^0_{vis}$, which appreciably simplifies the situation as discussed in Sec. 2.1.3. However, in general, the robot should consider that there may be parts of the environment that cannot be visited or seen and attempt the search only where it is possible. The *mobile search* problem is decomposed into the two stages consecutively solved on lines 2, 3. First, the *admissible region* is discretized by selecting a closed set of locations $\boldsymbol{l}^{set} \subset \mathcal{C}^0_{free}$ that cover it entirely; second, they are ordered into a sequence $\boldsymbol{l}^{seq}$ such that their consecutive visits yield an efficient search strategy. Since $\boldsymbol{l}^{seq}$ represents the search plan, the search execution is ready to start.

The outputs and covered region $\mathcal{W}_{cov}$ are initialized[1] on line 4. Next, the first location l_{goal} is popped (i.e., copied to a temporary variable and then removed) from the sequence, and then the robot starts following the shortest path towards l_{goal} (lines 6, 7). Meanwhile, in a loop (on lines 8-15) running with a fixed frequency f, the robot senses the environment (line 9), checks for the *target's* presence (lines 10-12), and updates its structures (lines 13, 14). If the *target* is detected (line 10), it is then localized (line 11), and the mission is successfully terminated (line 12).

[1] `NaN` stands for *not-a-number*, i.e., an invalid (or unknown) value.

Algorithm 3.2: *Mobile search*: simulation

Input: $\mathcal{W}$ ... environment, q_0 ... robot's initial configuration,
$r_{\mathcal{A}}$ ... robot's radius, $r_{\mathcal{V}}$ ... sensor's range, f ... sensing frequency
Output: t_{exp} ... expected time to find the searched object,
τ ... traversed trajectory, $\boldsymbol{l}^{true}$ ... truly visited sequence of locations

1 Compute $\mathcal{C}_{free}$, $\mathcal{C}^0_{free}$, $\mathcal{W}^0_{vis}$ structures.
2 $\boldsymbol{l}^{set} \leftarrow$ `Discretize`$(\mathcal{W}^0_{vis}, r_{\mathcal{V}}, \mathcal{C}^0_{free})$.
3 $\boldsymbol{l}^{seq} \leftarrow$ `Plan`$(\boldsymbol{l}^{set}, \mathcal{W}^0_{vis}, r_{\mathcal{V}})$.
4 $t_{exp} \leftarrow 0$, $t \leftarrow 0$, $\mathcal{W}_{cov} \leftarrow \emptyset$ ▷ `initialize`
5 **while** $\boldsymbol{l}^{seq}$ *is not empty* **do**
6 Pop the first location l_{goal} from $\boldsymbol{l}^{seq}$.
7 Start navigating the robot towards l_{goal} by the shortest path.
8 **repeat**
9 Get the current robot's configuration $q = (x, y, \phi)$.
10 Compute circle $\mathcal{C}_\circ(x, y, r_{\mathcal{V}})$ and *visibility polygon* $\mathcal{P}_{vis}(x, y)$.
11 $\mathcal{V} \leftarrow \mathcal{C}_\circ(x, y, r_{\mathcal{V}}) \cap \mathcal{P}_{vis}(x, y)$
12 $\mathcal{W}_{new} \leftarrow \mathcal{V} \setminus \mathcal{W}_{cov}$
13 $t_{exp} \leftarrow t_{exp} + t \cdot Area(\mathcal{W}_{new}) / Area(\mathcal{W}^0_{vis})$
14 Append (t, q) to the end of τ.
15 $\mathcal{W}_{cov} \leftarrow \mathcal{W}_{cov} \cup \mathcal{V}$
16 $t \leftarrow t + f^{-1}$
17 **until** l_{goal} *is reached*
18 Append l_{goal} to the end of $\boldsymbol{l}^{true}$.
19 $\boldsymbol{l}^{seq} \leftarrow$ `RePlan`$(\boldsymbol{l}^{seq}, \mathcal{W}^0_{vis}, r_{\mathcal{V}}, \mathcal{W}_{cov})$ ▷ `optional`
20 **return** t_{exp}, τ, $\boldsymbol{l}^{true}$

If the *target* remains undetected, the robot continues in the loop until it reaches the current goal l_{goal}. Then, optionally, the search plan can be updated (refined) based on the region $\mathcal{W}_{cov}$ that was covered so far on line 16 of the algorithm. Alternatively, the refinement can run parallelly with the sensing loop in a separate thread, and the plans (the current and the refined) can be just merged together on the line 16. Finally, the whole process (lines 5-16) repeats until the plan is exhausted. If the mission is not preliminarily interrupted by success, it ends in failure once the plan is empty, i.e., the whole environment was covered, and the object was not found.

Although Alg. 3.1 splendidly describes a realistic deployment of the *mobile search* in practice, it is not very convenient in regards to designing and tuning the planning algorithms encapsulated inside methods `Discretize`, `Plan`, and `RePlan`. The inconveniency rises from the fact that it does not directly return the expected (mean) time $t_{exp} = E[T \mid \tau]$ to find the searched object that those algorithms try to

minimize. If we wanted to obtain the expected time t_{exp}, we would have to execute lots of identical[2] experiments with the object placed randomly in the environment and then compute the mean of all recorded search times t_{search}. However, such methodology is neither precise, nor practical. Therefore, to simulate the search and test our algorithms efficiently, we use a different scheme that leans on the discretized definition (2.9) of the expected time.

The second scheme displayed in Alg. 3.2 has a similar structure to the previous one, except the search always unfolds until the search plan is exhausted, and the expected time to find the object is being progressively computed during the execution (lines 12, 13) and returned at the end. The other outputs of the algorithm are the trajectory τ traversed by the robot, and the truly visited sequence of goals $\boldsymbol{l}^{true}$. Note that if the replanning is not considered, then $\boldsymbol{l}^{seq}$ equals the initial sequence $\boldsymbol{l}^{seq}$ generated on line 3; else, the same does not hold. In this case, we work with the probability distribution of the object's location, so the object of interest does not need to appear in the simulation directly. Also, the simulation scheme does not consider readings from the sensor but instead directly computes the ideal version of $\mathcal{V}$ based on Eq. (2.11) (lines 9-11). This choice adds to our simulation's precision and saves us from implementing a transformation of sensory data into a polygonal domain.

To conclude, we approach the *mobile search* according to Alg. 3.2 because we can directly work with our problem's optimization criterion as defined in Eq. 2.9. The approach is equivalent to if we executed the search infinitely times according to Alg. 3.1 and then computed the mean time of the search. On the other hand, Alg. 3.1 may serve as instructions on how to use our solution in a real life scenario.

[2]By identical, we mean all the parameters (including all the random seeds except the one used for placing the object) of our implementation or the simulator would have to be the same.

3.2 Environment discretization

We encapsulate the planning algorithms for the *mobile search* into three functions: `Discretize`, `Plan`, and `RePlan`, as mentioned in the previous section. Functions `Discretize` and `Plan` correspond to the two stages of the decomposed problem, and `RePlan` is an optional procedure used for refining the plan during the execution. In this section, we discuss the *discretization*.

The *discretization* procedure takes the *admissible region for search* $\mathcal{W}^0_{vis}$, the visibility radius $r_\mathcal{V}$, and the collision-free subset of configuration space $\mathcal{C}^0_{free}$ as inputs and produces a closed set of reachable configurations $\boldsymbol{l} = \{l_i\}$, $l_i \in \mathcal{C}^0_{free}$. The resulting set must cover the whole *admissible region*, i.e.,

$$\bigcup_{l_i \in \boldsymbol{l}} \mathcal{V}(l_i) = \mathcal{W}^0_{vis}, \tag{3.1}$$

where $\mathcal{V}(l_i)$ is the *region sensed from* configuration l_i defined in Eq. (2.11). As stated in Sec. 2.1.3, we assume that any configuration $l_i \in \mathcal{C}^0_{free}$ is reachable from any other configuration $l_j \in \mathcal{C}^0_{free}$, the *sensed region* of the robot is independent of the robot's heading, and the robot is circular. Therefore, it is from now sufficient to see $\mathcal{C}^0_{free}$ and $\boldsymbol{l}$ as sets of reachable locations (2D points) and drop the third coordinate ϕ.

We show a general scheme of the *discretization* procedure in Alg. 3.3. The scheme is composed of two necessary steps performed by procedures `Cover` and `Fix`, respectively. Procedure `Cover` generates a specific set of polygons $\boldsymbol{\mathcal{R}}$ and locations $\boldsymbol{l}$. The sets must satisfy the following:

1. every polygon $\mathcal{R}_i \in \boldsymbol{\mathcal{R}}$ can be paired with a location $l_i \in \boldsymbol{l}$ such that the location is in its interior or on its border, i.e.,

$$l_i \in \mathcal{R}_i, \tag{3.2}$$

2. for every pair $(\mathcal{R}_i, l_i)$ all points of $\mathcal{R}_i$ are *visible* and within distance $r_\mathcal{V}$ from l_i, which is equivalent to $\mathcal{R}_i$ being a subset of the *region sensed from* l_i, i.e.,

$$\mathcal{R}_i \subset \mathcal{V}(l_i), \text{ and} \tag{3.3}$$

3. the polygons cover the whole *admissible region*, i.e.,

$$\bigcup_{\mathcal{R}_i \in \boldsymbol{\mathcal{R}}} \mathcal{R}_i = \mathcal{W}^0_{vis}. \tag{3.4}$$

Algorithm 3.3: General *discretization* procedure

```
1 Function Discretize(W^0_vis, r_V, C^0_free):
2     R, l ← Cover(W^0_vis, r_V)
3     R, l ← Filter(R, l)                          ▷ optional
4     R, l ← Fix(R, l, C^0_free)
5     l ← Improve(R, l)                            ▷ optional
6     l ← FilterVis(l, W^0_vis, r_V)               ▷ optional
7     return l
```

Note, that the common properties 3.3, 3.4 of sets $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$ are together equivalent to the single property 3.1 of $\boldsymbol{l}$.

After `Cover`, some points $l_i \in \boldsymbol{l}$ may still be unreachable for the robot, as the procedure does not consider $\mathcal{C}^0_{free}$. The reachability is enforced later by the other necessary step of the *discretization* — the `Fix` procedure. Procedure `Fix` modifies all pairs $(\mathcal{R}_i, l_i)$ such that l_i and all points of $\mathcal{R}_i$ are reachable while not breaking properties 3.1, 3.3. This is trivial for pairs which already satisfy reachability. For the other pairs this is not a trivial task in general. We do not in-depth study these cases but instead use for the task a simple algorithm without any guarantees regarding properties 3.1, 3.3. The simple `Fix` procedure shown in Alg. 3.4 seem to cause zero or insignificant inaccuracies based on our experiments. For each $\mathcal{R}_i \in \boldsymbol{\mathcal{R}}$ it computes an intersection with $\mathcal{C}^0_{free}$ (line 4) and if it's not empty (line 5), then the result of the intersection $\mathcal{R}'_i$ and a new location l'_i are added to new sets $\boldsymbol{\mathcal{R}'}$, $\boldsymbol{l'}$, respectively (line 10). The new location l'_i is the same as $l_i \in \boldsymbol{l}$, except if $l_i \notin \mathcal{R}'_i$, then arbitrary point from $\mathcal{R}'_i$ is taken instead (lines 6-9).

Besides the two necessary (`Cover` and `Fix`), there are other optional steps in Alg. 3.3: `Filter`, `Improve`, and `FilterVis`. These are meant to improve sets $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$ while preserving all their important properties discussed above. Filtering procedures, `Filter` and `FilterVis`, only reduce the sets (i.e., remove some unnecessary elements) and do not modify other elements in any way. Reducing sets $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$ speeds up other algorithms that work with them, and also, it may improve the final solution to the *mobile search*, which we demonstrate later in our experiments. `Filter` and `FilterVis` work similarly. The only difference is that `Filter` reduces based on the polygons from $\boldsymbol{\mathcal{R}}$ and `FilterVis` based on the *sensed regions* $\boldsymbol{\mathcal{V}} = \{\mathcal{V}(l_i)\}$, $\forall l_i \in \boldsymbol{l}$. Both methods are explained later in Sec. 3.2.3. Procedure `Improve` finds better placement of l_i in $\mathcal{R}_i$ according to a given criterion. It is only allowed to move around locations strictly inside their corresponding polygons, and it leaves the polygons unchanged.

Algorithm 3.4: Reachability enforcing procedure (no guarantees)

Function `Fix`($\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$, $\mathcal{C}^0_{free}$)**:**
  Initialize empty sets of polygons $\boldsymbol{\mathcal{R}'}$ and points $\boldsymbol{l'}$.
  foreach $(\mathcal{R}_i, l_i) \in \boldsymbol{\mathcal{R}} \times \boldsymbol{l}$ **do**
    $\mathcal{R}'_i \leftarrow \mathcal{R}_i \cap \mathcal{C}^0_{free}$
    if $\mathcal{R}'_i \neq \emptyset$ **then**
      if $l_i \in \mathcal{R}'_i$ **then**
        $l'_i \leftarrow l_i$
      else
        $l'_i \leftarrow$ Any point in $\mathcal{R}_i$.
      Add $\mathcal{R}'_i$, l'_i to sets $\boldsymbol{\mathcal{R}'}$, $\boldsymbol{l'}$, respectively.
  return $\boldsymbol{\mathcal{R}'}$, $\boldsymbol{l'}$

Regarding the quality of the generated locations, it is hard to say which properties would always yield good solutions to the *mobile search*, as discussed in Sec. 1.3.1. Here, our strategy is to adopt several approaches from the literature and also design our own, test all variants, and finally select the one which provides the best results to our original problem. All considered approaches differ primarily in the `Cover` procedure, while `Fix` and the filtering procedures stay the same for all. Furthermore, our approach relies on the `Improve` procedure, while methods adapted from the literature do not consider it. We discuss the literature's approaches, i.e., adapted versions of `Cover`, in the next Subsec. 3.2.1 and introduce our own, i.e., new `Cover` + `Improve` procedures, in Subsec. 3.2.2. The filtering schemes, i.e., `Filter` and `FilterVis`, applicable for all the considered approaches are described in detail in Subsec. 3.2.3. The last Subsec. 3.2.4 concerns a hybrid method combining our approach with one of the adapted from literature.

3.2.1 Literature: DT, KA, DS

We consider three implementations of the `Cover` procedure adapted from the literature characterized by the same types of inputs $\mathcal{W}^0_{vis}$, $r_{\mathcal{V}}$ and outputs $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$, whereas the output sets have properties (3.2), (3.3), (3.4), as explained previously.

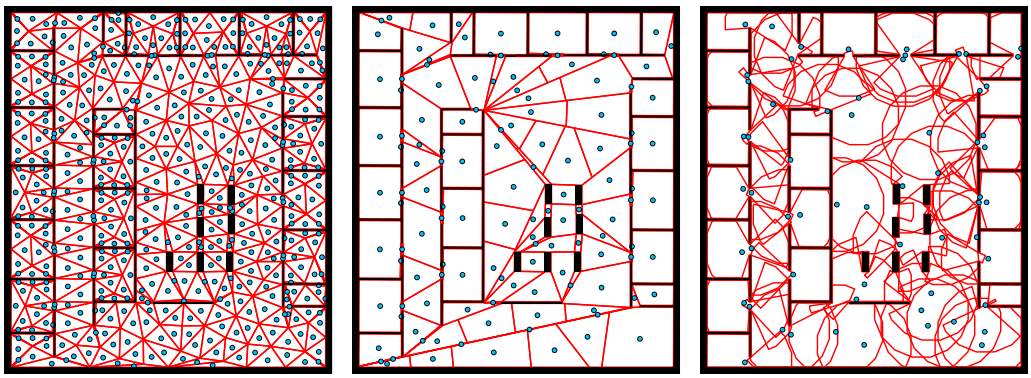
The first one relies on ***conforming constrained Delaunay triangulation* (DT)** [11] that creates a triangular mesh over the *admissible region* while constrained by several types of conditions, e.g., limiting the size of the triangles or minimal angle that appear in the mesh. In our case, the triangles correspond to the set of polygons $\boldsymbol{\mathcal{R}}$, and their centers (of mass) are taken as the locations $\boldsymbol{l}$.

They are constrained so that their circumscribed circles have radii smaller or equal to the visibility radius $r_{\mathcal{V}}$, and since they are convex, all their points are *visible* between each other. Therefore, the required conditions are satisfied and the method can be used.

The second `Cover` procedure by **Kazazakis and Argyros (KA)** [12] designed originally for the AGP with guards that have limited visibility, works similarly to the first one but is slightly more sophisticated. It first partitions the *admissible region* into a set of convex polygons of arbitrary size. The convex polygons are then sliced into smaller ones, which satisfy the visibility constraint, and for us, those make the $\boldsymbol{\mathcal{R}}$ set. Finally, the guards (locations) are so-called WS-points of the output polygons. The WS-point of a convex polygon is computed by averaging coordinates of centers of the polygon's edges weighted over the edges' lengths. It is easy to confirm that this method satisfies the required properties too. For more details, see the original work.

The third discretizing method is based on the idea of satisfying conditions 3.2, 3.3 in a trivial manner, i.e., l_i can be an arbitrary point in $\mathcal{W}_{vis}^0$ and $\mathcal{R}_i = \mathcal{V}(l_i)$. The essence of the method is adding polygons $\mathcal{V}(l_i)$ into the *admissible region* until it is wholly covered. This is done in a randomized fashion by the ***dual sampling* (DS)** [13] algorithm. The algorithm maintains a region $\mathcal{W}_{uncov}$ that was not yet covered (starting with $\mathcal{W}_{uncov} = \mathcal{W}_{vis}^0$) and randomly samples that region by pairs $(l_i, \mathcal{V}(l_i))$ which are always selected among candidates by a sub-procedure. When choosing a new i-th sample, the sub-procedure first picks an initial random point $l_i^0 \in \mathcal{W}_{uncov}$, and computes $\mathcal{V}(l_i^0)$. Then, just the polygon $\mathcal{V}(l_i^0)$ is further sampled by many temporary random points $\boldsymbol{l_i} = \{l_i^k\}$, $l_i^k \in \mathcal{V}(l_i^0)$: the candidates. Overall, only the candidate pair l_i^k, $\mathcal{V}(l_i^k)$ that maximizes the newly covered region $\mathcal{V}(l_i^k) \cap \mathcal{W}_{uncov}$ is picked as the new legitimate i-th sample l_i, $\mathcal{V}(l_i)$. Finally, the uncovered region is updated $\mathcal{W}_{uncov} \leftarrow \mathcal{W}_{uncov} \setminus \mathcal{V}(l_i)$ and the whole process repeats until $\mathcal{W}_{uncov}$ is empty, i.e., the whole *admissible region* is covered.

For a visual comparison of algorithms DT, KA, and DS, see Fig. 3.1. The methods solve the same instance of an environment that is about 33×37 meters large while assuming visibility radius $r_{\mathcal{V}} = 5\,\mathrm{m}$. Note that for DT and KA the polygons are disjunct, but for DS, they can overlap. The most apparent difference between DT and KA is in the density of the mesh.



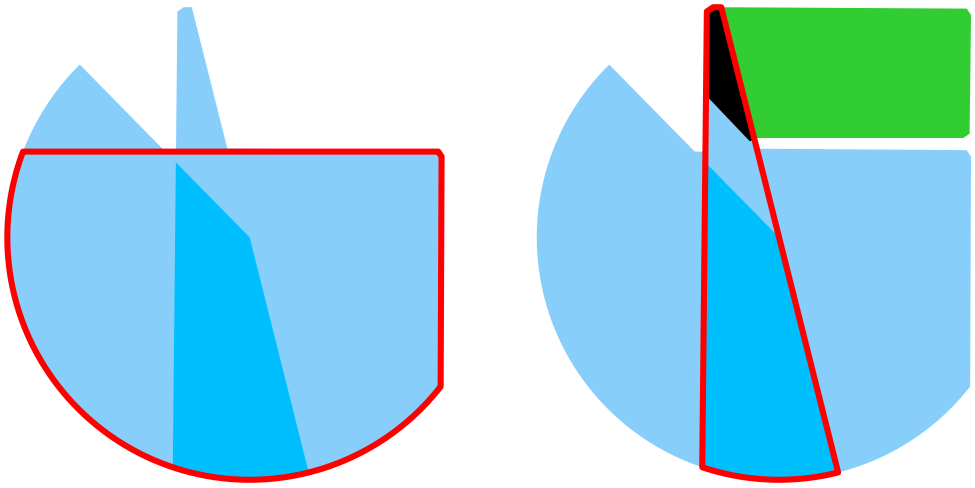
(a) : DT **(b) :** KA **(c) :** DS

Figure 3.1: Sets of polygons $\boldsymbol{\mathcal{R}}$ and points $\boldsymbol{l}$ produced by different `Cover` procedures adapted from the literature. Polygons' contours are shown in red, and the points are blue.

3.2.2 Proposed: WR

Our novel approach is motivated by utilizing a solution to the related **WRP problem (WR)** (discussed in Sec. 2.1.4). The WRP on the *admissible region* is solved (approximately) by decomposition to discretization and optimization similarly as we approach the whole *mobile search*. The solution to the WRP is a set of discrete cover-locations connected by a route whose length is the minimization criterion of the problem. The cover-locations are not fixed but can change their coordinates so that the covering condition (3.4) stays satisfied. This way, optimization is possible. The overall idea is to find the minimizing route represented as a sequence of locations connected by the shortest paths and declare those locations as our final set $\boldsymbol{l}$. The idea is realized by two procedures: `Cover`, and `Improve`.

First, let us review procedure `Cover`. We must generate such pairs $(\mathcal{R}_i, l_i)$ that regardless of how we move around l_i inside its polygon $\mathcal{R}_i$, the whole $\mathcal{R}_i$ always stays *visible* from l_i. Then, by fixing the polygons in place and allowing locations to move inside their paired polygons, we can perform optimization as discussed above and always be sure that the covering condition (3.4) holds. The properties required above yield that, in addition to condition (3.3), the polygons must also be convex and, and their farthest points must have maximal distance $r_{\mathcal{V}}$ from each other. To summarize, we look for a set of convex polygons that could fit inside a circle of radius $r_{\mathcal{V}}/2$ and cover the whole *admissible region*. To tackle the problem, we use a modified *dual sampling* algorithm, which instead of the whole polygons $\mathcal{V}(l_i)$ works with its subsets that are convex and half the radius. The convex subsets are created by cutting off parts of $\mathcal{V}_{1/2}(l_i) = \mathcal{C}_{\circ}(x_i, y_i, r_{\mathcal{V}}/2) \cap \mathcal{P}_{vis}(x_i, y_i)$ $(l_i = (x_i, y_i))$, until the result is a convex polygon. An important question here is how to select the proper cuts. One strategy is to aim for the maximal area of the result. This is equivalent to solving *maximum area convex subset* (MACS) problem.



(a) : MACS. **(b) :** MCCS.

Figure 3.2: A star-shaped polygon $\mathcal{V}$ and its kernel $\mathcal{K}$ are shown in light and bright blue color, respectively. The MACS of $\mathcal{V}$ is outlined in red in the left figure. The MCCS of $\mathcal{V}$, given the uncovered region $\mathcal{W}_{uncov}$, is outlined in red in the right figure. The region $\mathcal{W}_{uncov}$ is the union of black and green, and the intersection $\mathcal{V} \cap \mathcal{W}_{uncov}$ is just black.

Coeurjolly and Chasserythe [14] introduce an algorithm that approximately solves the problem for *star-shaped* polygons. Polygon is *star-shaped* if it has a specific subset called the *kernel* $\mathcal{K}$. For every point $p \in \mathcal{K}$ it applies, that all points of the original polygon are *visible* from p. It is easy to verify that polygon of our type, i.e., $\mathcal{V}_{1/2}(l_i)$, is *star-shaped* with minimal possible *kernel* $\mathcal{K} = \{l_i\}$. The algorithm by Coeurjolly and Chasserythe starts with a given *star-shaped* non-convex polygon. Then, according to several criteria, it determines a set of promising candidate cuts that would bring the polygon closer to convexity while preserving the kernel. Finally, from the candidate cuts, one that results in a maximal-area subset of the original polygon is selected and executed. This process repeats until a convex polygon is received. We adopt this procedure in our algorithm to transform $\mathcal{V}_{1/2}(l_i)$ into a convex polygon. However, instead of maximizing the area of the resulting polygon, we prefer cuts that maximize intersection with the uncovered region $\mathcal{W}_{uncov}$, which is maintained by the upper-level *dual sampling* algorithm. We call the result *maximally covering convex subset* (MCCS) of $\mathcal{V}_{1/2}(l_i)$. Examples of MACS and MCCS are shown in Fig. 3.2.

Finally, the modified *dual sampling* algorithm that utilizes MCCSs is shown in Alg. 3.5. The algorithm has one extra parameter, i_{max}, the number of *dual sampling* iterations, i.e., how many samples are considered for each polygon. Based on preliminary experiments, we use $i_{max} = 100$. In the algorithm, the output sets are first initialized as empty, and the *admissible region* $\mathcal{W}_{vis}^{0}$ is copied into the uncovered region $\mathcal{W}_{uncov}$ (lines 2, 3). Then, the main loop (lines 4-16), which runs

Algorithm 3.5: Proposed covering procedure (WR).

Parameters: i_{max} ... number of *dual sampling* iterations

1 **Function** `Cover`($\mathcal{W}^0_{vis}$, $r_\mathcal{V}$)**:**
2 Initialize empty sets of polygons $\boldsymbol{\mathcal{R}}$ and points $\boldsymbol{l}$.
3 $\mathcal{W}_{uncov} \leftarrow \mathcal{W}^0_{vis}$
4 **while** $\mathcal{W}_{uncov} \neq \emptyset$ **do**
5 Select a point $l_0 = (x_0, y_0) \in \mathcal{W}_{uncov}$ randomly.
6 Compute MCCS $\mathcal{R}_0$ of $\mathcal{C}_\circ(x_0, y_0, r_\mathcal{V}/2) \cap \mathcal{P}_{vis}(x_0, y_0)$.
7 $a_0 \leftarrow Area(\mathcal{R}_0 \cap \mathcal{W}_{uncov})$
8 $\mathcal{R}_{best}$, l_{best}, $a_{best} \leftarrow \mathcal{R}_0$, l_0, a_0
9 **for** $i \leftarrow 1, \ldots, i_{max}$ **do**
10 Select a point $l_i = (x_i, y_i) \in \mathcal{R}_0$ randomly.
11 Compute MCCS $\mathcal{R}_i$ of $\mathcal{C}_\circ(x_i, y_i, r_\mathcal{V}/2) \cap \mathcal{P}_{vis}(x_i, y_i)$.
12 $a_i \leftarrow Area(\mathcal{R}_i \cap \mathcal{W}_{uncov})$
13 **if** $a_i > a_{best}$ **then**
14 $\mathcal{R}_{best}$, l_{best}, $a_{best} \leftarrow \mathcal{R}_i$, l_i, a_i
15 Add $\mathcal{R}_{best}$, l_{best} to sets $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$, respectively.
16 $\mathcal{W}_{uncov} \leftarrow \mathcal{W}_{uncov} \setminus \mathcal{R}_{best}$
17 **return** $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$

until there is nothing left to cover, starts. Inside the loop, the initial sample l_0 is picked by random from $\mathcal{W}_{uncov}$, its MCCS $\mathcal{R}_0$ is determined, and the area a_0 of the newly covered region is computed (lines 5-7). The initial three entities are then copied to new variables, which hold the best sample so far (line 8). The inner fixed iteration loop follows (lines 9-14). The next three lines 10-12 are analogous to lines 5-7 (hence the name *dual sampling*), except the random samples are picked from $\mathcal{R}_0$ instead of $\mathcal{W}_{uncov}$. The best sample is updated in case the current area a_i is larger than the incumbent a_{best} (lines 13, 14). Outside the inner loop, the best sample is appended to partial solutions $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$, and the uncovered region $\mathcal{W}_{uncov}$ is updated (lines 15, 16). Finally, when the whole *admissible region* is covered, the completed sets $\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$ are returned (line 16).

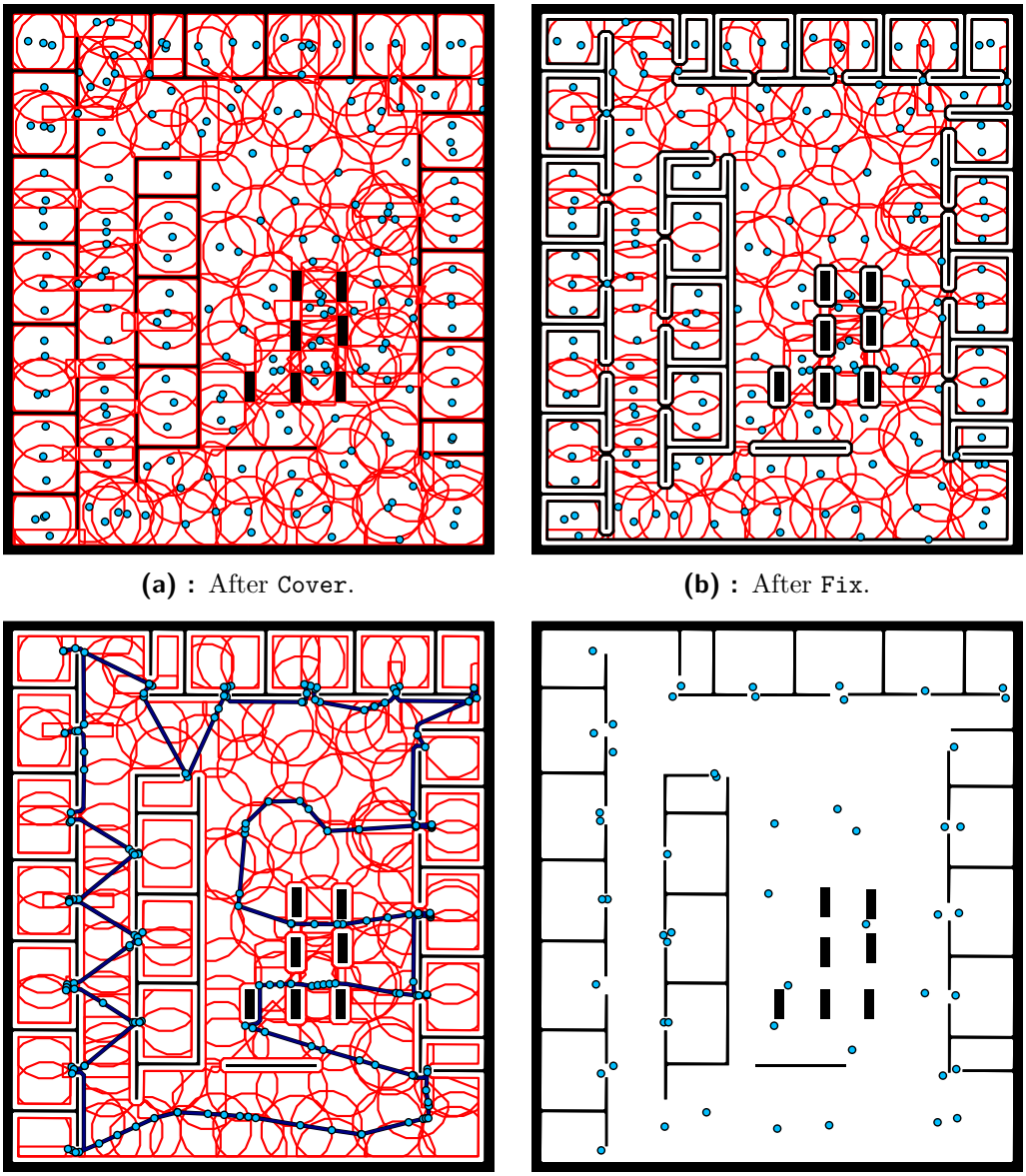
An example of resulting sets produced by the WR method is shown in Fig. 3.3a. The environment and visibility radius are the same as for methods DT, KA, DS in previous Fig. 3.1. The other subfigures of Fig. 3.3 display the *discretization* in different phases of the general scheme, Alg. 3.3. For example, Fig. 3.3b shows the points and polygons after a call to `Fix`, i.e., after removing parts of polygons that are not reachable by the robot and moving the points accordingly. Here, the considered robot's radius is $r_\mathcal{A} = 0.4\,\text{m}$, and the contours of $\mathcal{C}^0_{free}$ are shown in black.

Algorithm 3.6: Proposed improving procedure (WR)

```
1 Function Improve(𝓡, l):
2     Compute a distance matrix D = (d_{i,j}) of the shortest paths between all
        pairs of locations l_i, l_j ∈ l.
3     σ ← TSP(D)
4     l' ← TPPO(σ(𝓡), σ(l))
5     return l'.
```

After `Fix`, the *discretization* might end, and the set of points would be final. However, as advertized before, the WR method includes an extra optimization phase encapsulated into the `Improve` method that we detail next. In the optimization part, we wish to move the cover-locations closer to each other such that visiting them all takes lesser time. Also, if we consider an environment such as an office with rooms each having a single entrance, we like to have locations placed nearby the entrances, as can be seen in the example in Fig. 3.3d. To achieve this, the WR method fixes in place the polygons generated by Alg. 3.5, then the polygons are put to a sequence in some reasonable order, and finally, having the order fixed, the *touring polygons problem* (TPP) [16] is solved. The TPP looks for the minimal length cyclic tour that visits all polygons in the sequence. The optimization is performed by shifting the points inside their corresponding polygons until the solution converges to the minimal length. Let the sequence of polygons be $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$. The TPP tour is represented as a sequence of points $(p_1, \ldots, p_n)$, where the i-th point p_i visits i-th polygon $\mathcal{R}_i$, i.e., $p_i \in \mathcal{R}_i$. The original problem does not consider any obstacles, so the tour points are joined with straight line segments. However, in our case where the obstacles are present, the shortest collision-free paths connect the tour points, and we formally denote such modified problem as *TPP with obstacles* (TPPO).

The scheme of the WR `Improve` method is shown in Alg. 3.6. The ordering σ is determined by a solution of the TSP with distances as the lengths of the shortest paths between all pairs of the input locations $\boldsymbol{l}$ (lines 2, 3). For the TSP, we use Helsgaun's effective implementation of the *Lin-Kernighan heuristic* (LK), called LKH [80], coded as the *LKH-3* [81] program available at `http://webhotel4.ruc.dk/~keld/research/LKH-3/`. The TPPO is solved as part of Vidašič's diploma thesis [82] studying the *TSP with neighborhoods*. We use the author's solution (algorithm and implementation) to approximately solve the problem in our WR `Improve` method (line 4). The output representation of the TPPO route is then desequenced, i.e., transformed to a set of points, and returned by `Improve` as the cover-locations set (line 5). All possible TPPO outputs are guaranteed to cover the whole *admissible region* thanks to the properties of the input polygons $\boldsymbol{\mathcal{R}}$ generated by the WR `Cover` method (Alg. 3.5), as explained before.



(a) : After `Cover`.

(b) : After `Fix`.

(c) : After `Improve`.

(d) : After `FilterVis`.

Figure 3.3: The proposed approach (WR) to *discretization* displayed in different phases of the general procedure introduced in Alg. 3.3.

An example of a solution to the WRP achieved by combining Alg. 3.5 and 3.6 is shown in Fig. 3.3c, in dark blue color. The locations that represent the route might be returned by the *discretization* (Alg. 3.3) as final, or one additional step might be performed — the filtering. An example of filtered locations are shown in Fig 3.3d, and the next subsection explains the related algorithms.

Algorithm 3.7: Filtering procedures

1 **Function** `Filter`($\boldsymbol{\mathcal{R}}$, $\boldsymbol{l}$)**:**
2 Let $n = |\boldsymbol{\mathcal{R}}| = |\boldsymbol{l}|$ and sort sets $\boldsymbol{\mathcal{R}} = (\mathcal{R}_k)$, $\boldsymbol{l} = (l_k)$, $k = 1, \ldots, n$ in ascending order according to the polygons' areas.
3 Create a set of indices $\boldsymbol{j} = \{1, \ldots, n\}$.
4 Initialize empty sets of polygons $\boldsymbol{\mathcal{R}'}$ and points $\boldsymbol{l'}$.
5 **for** $i \leftarrow 1, \ldots, n$ **do**
6 $\mathcal{R}_{union} \leftarrow \bigcup_{j \in \boldsymbol{j}, j \neq i} \mathcal{R}_j$
7 **if** $\mathcal{R}_i \setminus \mathcal{R}_{union} = \emptyset$ **then**
8 Remove i from $\boldsymbol{j}$.
9 **else**
10 Add $\mathcal{R}_i$, l_i to sets $\boldsymbol{\mathcal{R}'}$, $\boldsymbol{l'}$, respectively.
11 **return** $\boldsymbol{\mathcal{R}'}$, $\boldsymbol{l'}$
12 **Function** `FilterVis`($\boldsymbol{l}$, $\mathcal{W}^0_{vis}$, $r_{\mathcal{V}}$)**:**
13 Create a new set $\boldsymbol{\mathcal{V}} = \{\mathcal{V}_i\}$, where $\mathcal{V}_i = \mathcal{C}_\circ(x_i, y_i, r_{\mathcal{V}}) \cap \mathcal{P}_{vis}(x_i, y_i)$ for each $l_i = (x_i, y_i) \in \boldsymbol{l}$.
14 $\boldsymbol{\mathcal{V}}_{unused}, \boldsymbol{l} \leftarrow$ `Filter`$(\boldsymbol{\mathcal{V}}, \boldsymbol{l})$
15 **return** $\boldsymbol{l}$

3.2.3 Location filtering

Location filtering systematically removes some locations from set $\boldsymbol{l}$ so that the remaining are still completely covering. The *discretization* in Alg. 3.3 considers two filtering methods: `Filter` and `FilterVis`, both detailed in Alg. 3.7. `Filter` (lines 1-11) considers both points $\boldsymbol{l}$ and polygons $\boldsymbol{\mathcal{R}}$ as inputs/outputs. `FilterVis` (lines 12-15) is just a special case of the previous with pre-defined polygons $\boldsymbol{\mathcal{R}} = \boldsymbol{\mathcal{V}}$ as the *regions* defined in Eq. (2.11) *sensed from* the input locations $\boldsymbol{l}$.

In procedure `Filter`, the input sets are first sorted in ascending order according to the polygons' areas (line 2). Next, a set of the remaining indices $\boldsymbol{j}$ and the output sets are created and initialized (lines 3, 4). Each polygon $\mathcal{R}_i$ (from the sorted input set) is then investigated in a loop (lines 5-10). Inside the loop, the union $\mathcal{R}_{union}$ of other remaining polygons is computed (line 6). If the difference between $\mathcal{R}_i$ and $\mathcal{R}_{union}$ is empty, then $\mathcal{R}_i$ is redundant and index i can be removed from $\boldsymbol{j}$ (lines 7, 8). If else, then the i-th polygon and location are added to the output sets (lines 9, 10), which are returned at the end (line 11).

Filtering can be without a difference combined with either one of the considered methods: DT, KA, DS, WR. We call the methods with filtering DTF, KAF, DSF, WRF, respectively.

3.2.4 Hybrid: WRF-DT-F

Preliminary tests and visualizations of the output cover-locations have shown that DTF and WRF might produce the best discretizations with respect to the *mobile search*. Here, we introduce a hybrid approach combining both methods' strengths that can produce even better results. We call the hybrid method WRF-DT-F, and it considers two types of cover-locations: first $\boldsymbol{l_1}$ produced by WRF, and second $\boldsymbol{l_2}$ by DT. These two sets of different types are united into a single set, i.e., $\boldsymbol{l} = \boldsymbol{l_1} \cup \boldsymbol{l_2}$, final filtering by `FilterVis` is performed, and the ultimate result is returned.

3.3 Metaheuristic for the TDP

This section concerns the other major part of the *mobile search* solution — *the routing* — realized by the `Plan` procedure in the general search scheme, Alg. 3.2. More specifically, it proposes a metaheuristic for the TDP, the most simplified version of the main problem. It assumes that the sensing of the environment is performed exclusively on the cover-locations and that all locations have static equal information gain. Although these assumptions are not realistic, it is essential to consider the TDP as a well-studied problem, for which exist benchmark instances and reference metaheuristics. We extend it to model the *mobile search* better later in this work.

An introduction to existing metaheuristics for the TDP and similar problems is given in Sec 1.3.3. In the process of designing the novel metaheuristic, we focus on GVNS and GRASP and their main components (e.g., local search, *perturbation*, stopping conditions) and create variants that generally differ in the points that follow.

1. General scheme of the method:
   a. GVNS,
   b. GRASP,
   c. a combination of the two, i.e., GRASP that uses GVNS instead of VND in local search.
2. The sequence of neighborhoods used within perturbation (applicable only to (1a) and (1c)).
3. The constructive heuristic (applicable only to (1b) and (1c)):
   a. deterministic,
   b. randomized with a fixed rate of randomness,
   c. randomized with a randomly chosen rate of randomness.
4. Stopping condition of the inner heuristic (applicable only to (1c)):
   a. stop after a fixed number of iterations j_{max},
   b. stop after j_{max} iterations without any further improvement.
5. The value of j_{max} constant (applicable only to (1c)).
6. Variant of VND:
   a. fixed-sequence basic VND (the classical one),
   b. random-sequence basic VND.

7. The set of neighborhoods used within VND.

8. Order of the neighborhoods (applicable only to (6a)).

Each point on the list is detailed in later subsections. The novel metaheuristic is to be systematically designed and experimentally evaluated against the current state-of-the-art method (the reference) using the RTD methodology described in Sec. 1.3.4 and 1.3.5. The choice of the reference is done and explained in Sec. 3.3.1. A detailed description of the considered algorithms, their variants, and technical details in Sec. 3.3.2-3.3.8. We provide a handy overview of symbols that appear in Sec. 3.3.2-3.3.8 in Table 3.1. The union of symbols in the table and the symbols defined in Sec. 2.2 creates a full set of special symbols used in the algorithms' descriptions. Any other symbols denote temporal variables created within the algorithm.

3.3.1 Reference: GILS-RVND

As our reference - the state-of-the-art method for the TDP - we choose the original GILS-RVND [30] for its simplicity and the right trade-off between solution quality and run-time. We purposely leave out the parallel versions proposed by [73] since we assume single-processor computing. Although it genuinely raises the bar for the traditional solving of the TDP, we also decide not to consider the newest improved version MDM-GILS-RVND [74]. The reasons for this decision we explain next. The MDM improvement over the original is more significant as the instances go large, e.g., up to 500 or 1000 customers. However, for these instances, the reported run times (> 500 seconds) are still a lot above the range that our thesis mainly focuses on (< 100 seconds). Since, in the first half of its iterations, the MDM version is practically identical to its predecessor, it is reasonable to assume that on the instance which takes MDM, e.g., 500 seconds to solve, after 100 seconds of run-time, the average quality of the incumbent solution would be no better than in case of plain GILS-RVND. That being said, we choose the simpler of the two algorithms while obtaining nearly equivalent comparison with our method as if we have chosen the more complex one. Also, the MDM extension to GILS-RVND introduced by [74] is a general one and might be applied only with minor adjustments to any greedy or semi-greedy restarting heuristic solving any TSP or TDP variant, where the solution is a Hamiltonian path or cycle. As future research, we also consider extending our proposed method to the MDM version. For now, nevertheless, we regard heuristics without DM techniques as they bring no additional value for scenarios we study.

3.3.2 Stopping conditions

We consider three general schemes: GVNS (1a), GRASP (1b), and a combination of the previous two GRASP-GVNS (1c). The schemes accept some common inputs and return a valid solution. The common inputs have a connection with the stopping condition of the algorithms and we denote their tuple as $in = (i_{max}, c_{goal}, t_{max})$, where i_{max} is the number of main-loop iterations, c_{goal} is the target solution cost, and t_{max} is the CPU time limit.

Given the constant i_{max}, the algorithm stops and returns a valid solution after a fixed number of iterations i_{max}, if not stopped earlier by other criteria. Given the CPU time limit, the algorithm finishes, at worst, after t_{max} seconds. Of course, we can expect the real run-time to be a little higher than t_{max}, as the time limit check frequency is finite, and the algorithm always needs to finish certain operations after the exceedance is detected. However, our algorithms for the TDP are designed so that the caused delay is negligible for the robotic applications. We must note that the TDP is solved in a mission planning unit of a robot, which is not safety-critical. At last, the algorithm can also stop after it has found a solution with cost smaller or equal to given goal c_{goal}.

All considered algorithms are expected to run in several different modes depending on the combination of stopping conditions. For instance, when i_{max} is some positive integer, $t_{max} = \infty$, and $c_{goal} < 0$, the algorithm will always stop after the fixed number of iterations i_{max}. This configuration is the most common in the literature. Some other possibility is, e.g., to set $i_{max} = \infty$, and t_{max}, c_{goal} to some reasonable values, and the algorithm will either stop after it has found a good enough solution or after the time limit has passed. In this configuration, the algorithm fits a practical version of *Las Vegas improving algorithm* with computational time limit as defined in Sec. 1.3.5. The introduced variability opens a range of different applications and a possibility to generate several types of results used to compare the algorithms in various scenarios.

3.3.3 General schemes

The scheme of GVNS is presented in Algorithm 3.8. In the initialization (line 2-3), an iteration counter i is set to 1, a stopping flag *stop* is set to `false`, and an initial solution is constructed and assigned to $\mathcal{X}^\star$ variable, which holds the incumbent (the best overall) solution. The number 1 passed as an argument to the `Construct` procedure corresponds to a deterministic greedy construction (3a). The main loop (lines 4-14) runs until the *stop* flag is `true` or the maximum number of iterations

Symbol	Meaning
i_{max}	The number of main-loop iterations.
c_{goal}	Target solution cost.
t_{max}	CPU time limit.
j_{max}	The number of inner-loop iterations.
s_{rcl}	A size of a Restricted Candidate List (RCL).
$\mathcal{R}$	A set of $\lvert\mathcal{R}\rvert$ real $[0, 1]$-interval values $\{ r_1, r_2, \dots \}$.
$\mathcal{P}$	A sequence of $\lvert\mathcal{P}\rvert$ positive integers $(p_1, p_2, \dots)$.
$\boldsymbol{\mathcal{N}}$	A sequence of $\lvert\boldsymbol{\mathcal{N}}\rvert$ neighborhoods $(\mathcal{N}_1, \mathcal{N}_2, \dots)$.

Table 3.1: Overview of symbols that appear in algorithms related to the TDP.

Algorithm 3.8: *General variable neighborhood search* (GVNS)

```
1  Function GVNS(i_max, c_goal, t_max, P, N):
2      X* ← Construct(1)
3      i ← 1; stop ← false
4      while stop = false and i ≤ i_max do                ▷ main GVNS loop
5          k ← 1
6          while stop = false and k ≤ |P| do              ▷ inner GVNS loop
7              X ← Shake(X*, p_k)
8              X, stop ← Improve(X, t_max, c_goal, N)
9              if cost(X) < cost(X*) then
10                 X* ← X
11                 k ← 1
12             else
13                 k ← k + 1
14         i ← i + 1
15     return X*
```

i_{max} is reached. Next, index k is initialized to one (line 5), and the inner loop follows (lines 6-13). The algorithm is parameterized by a sequence of positive integers $\mathcal{P} = (p_1, p_2, \dots)$, which take a role in the *perturbation* phase (line 7), where the k-th member of the sequence is passed to the `Shake` procedure. The procedure is applied to the incumbent and results in a new current solution $\mathcal{X}$, which is improved (line 8) and evaluated (lines 9-13). If the cost of the current is less than the cost of the incumbent solution, then the current is assigned to the incumbent, and k is reset back to 1. Else, k is incremented, and the loop starts over with the next parameter in $\mathcal{P}$. Note the *stop* flag returned from the improving procedure (line 8). If it is true, then GVNS quickly comes to an end and returns the incumbent solution. If not stopped by the flag, the algorithm performs i_{max} iterations, terminates and returns the incumbent.

Algorithm 3.9: *Greedy randomized adaptive search procedure* (GRASP)

```
1  Function GRASP(i_max, c_goal, t_max, R, N):
2      i ← 1; stop ← false; c* ← ∞
3      while stop = false and i ≤ i_max do                ▷ main GRASP loop
4          α ← random value ∈ R
5          s_rcl ← max(1, ⌊α · N⌋)
6          X ← Construct(s_rcl)
7          X, stop ← Improve(X, t_max, c_goal, N)
8          if cost(X) < c* then
9              X* ← X
10             c* ← cost(X)
11         i ← i + 1
12     return X*
```

The scheme of GRASP is shown in Algorithm 3.9. Most of its parts are analogous to GVNS, except the construction procedure is included within the iteration loop (lines 4-6), and the *perturbation* phase, together with the whole inner loop, is omitted. In GRASP, a new solution is constructed in each iteration, then improved and evaluated. The construction is done in a greedy randomized fashion where the integer parameter s_{rcl} controls the level of randomness. Admissible values for s_{rcl} are in a range from 1 to N, where N is the size of the instance. 1 corresponds to a purely greedy solution and N to a totally random solution. s_{rcl} can be either set to some fixed value from the range, or can be constructed as on lines 4-5 of the GRASP algorithm. The latter option enables s_{rcl} to vary in each iteration.

Both schemes are combined in Algorithm 3.10. Line 7 of Alg. 3.9 is replaced by lines 4-14 of Alg. 3.8. Some variables are renamed to prevent ambiguity. In GRASP-GVNS (G+G) a new solution is constructed in each iteration (line 5-6) as in GRASP, and then it is systematically being perturbed (line 11), improved (line 12), and evaluated (lines 13-18) as in GVNS. After the whole GVNS procedure (lines 7-19) ends and returns the current solution $\mathcal{X}$, the final evaluation (lines 20-22) finishes the current iteration as in GRASP. Within one iteration of G+G, several GVNS iterations are performed. The number of GVNS iterations is given by an extra parameter j_{max}. The line 16 of Alg. 3.10 is optional, and it can be either omitted (4a), or not (4b). If omitted, then j_{max} is the exact number of GVNS iterations. If the line is present, then the GVNS iteration counter j resets to 1 with each improvement, and the GVNS loop only breaks after j_{max} iterations with no observed improvement.

Algorithm 3.10: GRASP-GVNS (G+G)

```
1  Function G+G(i_max, c_goal, t_max, j_max, R, P, N):
2      i ← 1; stop ← false; c* ← ∞
3      while stop = false and i ≤ i_max do                    ▷ main GRASP loop
4          α ← random value ∈ R
5          s_rcl ← max(1, ⌊α · N⌋)
6          X ← Construct(s_rcl)
7          j ← 1
8          while stop = false and j ≤ j_max do                ▷ main GVNS loop
9              k ← 1
10             while stop = false and k ≤ |P| do              ▷ inner GVNS loop
11                 X' ← Shake(X, p_k)
12                 X', stop ← Improve(X', t_max, c_goal, N)
13                 if cost(X') < cost(X) then
14                     X ← X'
15                     k ← 1
16                     j ← 1                                  ▷ only for (4b)
17                 else
18                     k ← k + 1
19             j ← j + 1
20         if cost(X) < c* then
21             X* ← X
22             c* ← cost(X)
23         i ← i + 1
24     return X*
```

3.3.4 Construction

The `Construct` procedure implements Greedy Randomized Adaptive (GRA) construction shown in Algorithm 3.11. First, a partial solution $\mathcal{X}$ is initialized with the depot s and Candidate List (CL) with the remaining vertices (line 2). In the main loop (lines 3-7), Restricted Candidate List (RCL) is built by considering only $\min(s_{rcl}, |\text{CL}|)$ nearest CL elements with respect to the last added vertex to $\mathcal{X}$ (line 4). $s_{rcl} \in \{1, \ldots, N\}$ is an argument passed to the procedure and $|\text{CL}|$ is the cardinality of CL. Finally, a candidate is selected from RCL by random, appended to the end of $\mathcal{X}$, and removed from CL (lines 5-7). The process repeats until CL becomes empty, i.e., all vertices are added to $\mathcal{X}$, and then the finished solution is returned.

Algorithm 3.11: *Greedy randomized adaptive* (GRA) construction

```
1 Function Construct(s_rcl):
2     x ← s; 𝒳 ← (x); CL ← V \ {x}
3     while CL is not empty do
4         Create a set RCL ⊂ CL considering only min(s_rcl, |CL|) nearest
           candidates to x.
5         x ← random value ∈ RCL
6         Append x to the end of 𝒳.
7         CL ← CL \ {x}
8     return 𝒳
```

Algorithm 3.12: *Perturbation* procedure

```
1  Function Shake(𝒳, p):
2      p ← min(p + 1, |𝒳|) − 1                    ▷ adjust p in case |𝒳| < p
3      Create p + 1 random subpaths 𝒮_0, 𝒮_1, ..., 𝒮_p of 𝒳, where 𝒮_0 is the one
        starting with the depot. This can be done by removing p random
        distinct edges from 𝒳.
4      Create sequence of indices ℐ = ( 1, 2, ..., p ) and shuffle it randomly.
5      𝒳' ← 𝒮_0
6      foreach i ∈ ℐ do
7          flip ← random Boolean value (true or false)
8          if flip then
9              Append reversed 𝒮_i to the end of 𝒳'.
10         else
11             Append 𝒮_i to the end of 𝒳'.
12     return 𝒳'
```

We consider three variants of the construction procedure in accordance with Section 1.1, Point (3) on the list: a deterministic (3a), randomized with a fixed level of randomness (3b), and randomized with a rate of randomness randomly chosen from a uniform discrete probability distribution (3c). All variants can be implemented by the `Construct(`s_{rcl}`)` procedure, where $s_{rcl} = 1$ for (3a), s_{rcl} is fixed in the range from 2 to N for (3b), and s_{rcl} is constructed as in lines 4-5 of Alg. 3.9 for (3c).

3.3.5 Perturbation

The basic VNS employs a mechanism that prevents the metaheuristic from getting stuck in local optima. Mladenović and Hansen [42] call this mechanism *shaking* or the *shake phase* in the original paper where the VNS was introduced. The *shaking* is also present in the VNS's generalized version that we use, and we also refer to it by the term *perturbation*. It resembles the work of Silva et al. [30], who use the *perturbation* called *double-bridge* for the TDP. *Double-bridge* was originally developed by Martin et al. [83] for the TSP. It removes and re-inserts four edges from and to the given path such that a new feasible path is generated. The edges to be removed are chosen randomly, and the way the path is glued back together is randomized as well. The procedure `Shake` shown in Algorithm 3.12 generalizes the mechanism by considering p edges instead of four.

The *perturbation* used in the *shaking* of GVNS works as follows. A valid path $\mathcal{X}$ and a positive integer parameter p are passed to the procedure. First, the path is partitioned by removing p random distinct edges from it (line 3). The subpaths resulting from this operation are labeled as $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_p$ in the order they appear in $\mathcal{X}$. Next, the first subpath $\mathcal{S}_0$ is assigned to a partial solution $\mathcal{X}'$ (line 5), and a sequence $\mathcal{I}$ of indices from 1 to p is created and randomly shuffled (line 4). The algorithm then goes through each index i in the randomized sequence (lines 6-11) and appends either the corresponding $\mathcal{S}_i$ (line 11) or its reversed version (line 9) to the end of $\mathcal{X}'$. The chance of reversing $\mathcal{S}_i$ before appending it to $\mathcal{X}'$ is 50% (line 7). The partial solution $\mathcal{X}'$ becomes feasible after the last remaining subpath is appended to it, and at this point, the procedure ends and returns $\mathcal{X}'$. Note that `Shake(`$\mathcal{X}$`, 4)` corresponds to the original *double-bridge*.

3.3.6 Local search

The local search in all considered general schemes (Alg. 3.8-3.10) is performed by a method based on *variable neighborhood descent* (VND). VND explores a solution space using several neighborhood structures. Its success relies on the following facts: a local optimum for one neighborhood structure is not necessarily a local optimum with respect to another neighborhood structure, and a global optimum is a local optimum with respect to all considered neighborhood structures. Mjirda et al. [84] provide an overview of sequential VND variants and their comparison on the TSP. With respect to their notion, we use *basic VND* with *the best improvement strategy*, a variant that performed the best in combination with the GVNS scheme, as the authors report.

Algorithm 3.13:
Variable neighborhood descent (VND) and *Randomized VND* (RVND)

```
1  Function Improve(𝒳, t_max, c_goal, 𝒩):
2      i ← 1; stop ← false
3      Shuffle sequence 𝒩 randomly.                    ▷ only for RVND (6b)
4      while i ≤ |𝒩| do
5          Denote the i-th neighborhood structure in sequence 𝒩 as 𝒩_i.
6          𝒳' ← arg min_{x̃ ∈ 𝒩_i(𝒳)} cost(x̃)
7          if cost(𝒳') < cost(𝒳) then
8              𝒳 ← 𝒳'
9              i ← 1
10             if cost(𝒳) ≤ c_goal then
11                 stop ← true
12                 break
13             Shuffle sequence 𝒩 randomly.            ▷ only for RVND (6b)
14         else
15             i ← i + 1
16         Get the total CPU time t since start.
17         if t ≥ t_max then
18             stop ← true
19             break
20     return 𝒳, stop
```

Additionally, the order in which the neighborhoods are considered in VND can be either fixed (deterministic) (6a) or randomized (6b). The latter, *randomized VND* (RVND), randomly selects an available neighborhood to be used in each iteration. Satyananda and Wahyuningsih [85] compare the performance of VND and RVND on instances of the *capacitated vehicle routing problem*. Here, the selection of operators in random order outperformed the fixed-sequence VND in a matter of solution quality, however the classical VND usually required lesser iterations to reach the local optimum. Similar observations report [30] for the TDP after obtaining some preliminary results. Nevertheless, neither [85] nor [30] consider real-time application scenarios as we do. In our context, the supremacy of RVND over VND is not so obvious, especially if the lesser iterations of VND and the complex search strategies (Alg. 3.8, 3.9, 3.10) are considered. Thus, we study both variants in this thesis.

The pseudo-code of (R)VND is shown in Algorithm 3.13. The method takes as an input an initial solution, which is about to be modified to improve its quality and returned in the end. Alternatively, no improvement can be found, and the same solution as the initial one is returned. The method also checks for the possible accomplishment of given goals. One of the goals is met, when the total run-time since the start of the most superior scheme is over the given value t_{max}. The other goal is met, when (R)VND finds a solution with cost c such that $c \leq c_{goal}$. If at least one of these two situations is detected, the method terminates immediately and returns the best solution found so far together with a flag *stop* indicating the accomplishment of the goals. In addition, the procedure is parametrized by a sequence $\boldsymbol{\mathcal{N}} = (\mathcal{N}_1, \mathcal{N}_2 \dots)$ of operators also called neighborhood structures. In general, $\mathcal{N} \in \boldsymbol{\mathcal{N}}$ is an operator, which takes a feasible solution $\mathcal{X} \in \mathcal{H}(\pi)$ of an instance π and a tuple of parameters as an input and returns a new feasible solution $\mathcal{X}' \in \mathcal{H}(\pi)$ as the output. The range of solutions that can possibly be obtained by applying an operator $\mathcal{N}$ on a particular solution $\mathcal{X}$ is called the $\mathcal{N}$-neighborhood of $\mathcal{X}$ and is denoted as $\mathcal{N}(\mathcal{X})$.

The initialization of Alg. 3.13 is done first. A neighborhood structures counter i is set to 1, the *stop* flag is set to `false` (line 2), and just in case of RVND, the members of the sequence $\boldsymbol{\mathcal{N}}$ are randomly shuffled (line 3). The main loop follows (lines 4-19). Inside, the best neighbor solution $\mathcal{X}'$ of neighborhood $\mathcal{N}_i(\mathcal{X})$ is found (line 6). The neighborhood structure $\mathcal{N}_i$ is the i-th of the sequence $\boldsymbol{\mathcal{N}}$ and $\mathcal{X}$ is the currently best solution. If the cost of $\mathcal{X}'$ is less then the cost of $\mathcal{X}$ (line 7), then $\mathcal{X}'$ is assigned to $\mathcal{X}$, counter i is set back to 1 (line 9), and in case of RVND, the sequence $\boldsymbol{\mathcal{N}}$ is again shuffled (line 13). In addition, the cost goal check is performed (lines 10-12), with the chance of breaking the main loop and setting the *stop* flag to `true` if the goal-accomplishment condition is satisfied. If the solution improvement condition on line 7 is not satisfied, then the counter i is raised up by one (line 15), which assures that the next neighborhood structure in the sequence is selected in the next run of the main loop. At last, the run-time goal check is performed (lines 16-18). First, the total CPU time t is obtained (line 16), and then the check is done in analogous way as in case of the cost goal check (lines 17-18). The total time t can be computed as $t = t_{current} - t_{start}$, where $t_{current}$ is the current CPU time and t_{start} is the CPU time of the start of the most superior scheme from which the `Improve` procedure was called. Depending on the general scheme considered with the (R)VND, the value of t_{start} is saved to memory at the first line of Alg. 3.8, 3.9, or 3.10. The main loop ends after all available neighborhood structures have been tried, and no more improvement was obtained, or it can be terminated prematurely by the accomplishments of the goals.

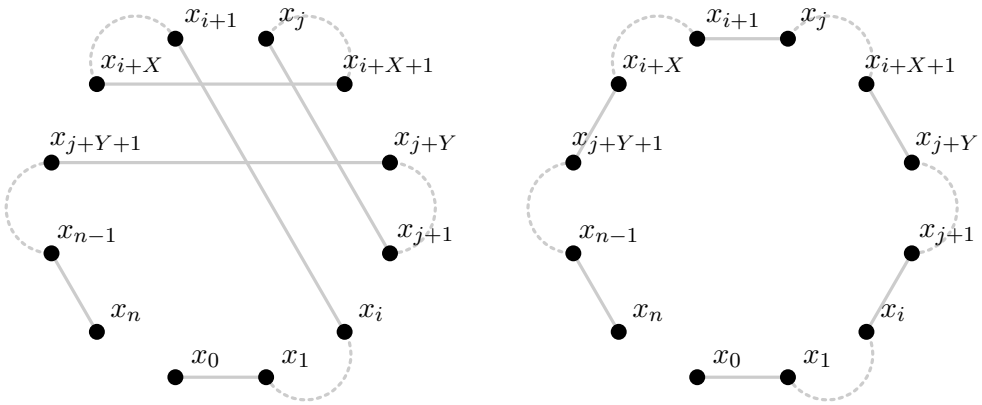


Figure 3.4: General operator `2-string`$(\mathcal{X}, X, Y, i, j)$ applied on path $\mathcal{X} = (x_k)$ for $k = 0, \ldots, n$, where inequalities $0 < X$, $0 < Y$, $i + X < j$, and $j + Y < n$ hold. Edges (x_i, x_{i+1}), (x_{i+X}, x_{i+X+1}), (x_j, x_{j+1}), (x_{j+Y}, x_{j+Y+1}) are removed and replaced by edges (x_i, x_{j+1}), (x_{j+Y}, x_{i+X+1}), (x_j, x_{i+1}), (x_{i+X}, x_{j+Y+1}). Left: the original path $\mathcal{X}$, right: the resulting path $\mathcal{X}'$ obtained by applying the operator on $\mathcal{X}$.

3.3.7 Local search operators

The core of the improvement procedure described in the previous subsection is a systematic exploration of a solution space using several neighborhood structures (operators). Here, we consider operators which are often used for solving TSP, TDP, and other VRPs. The complete set is as follows: `2-opt`, `1-point`, `or-opt2`, `or-opt3`, `or-opt4`, `or-opt5`, `2-point`, and `3-point`. Operator `2-opt` takes two non-adjacent edges from path $\mathcal{X}$ and replaces them by two new edges in order to obtain a new feasible path $\mathcal{X}'$. All the other operators, unlike `2-opt`, can be defined as a special case of more general operator that we call `2-string`. The definition of `2-string` comes next. Let $\mathcal{Y}$ be a set of all tuples $(\mathcal{X}, X, Y, i, j)$ such that $\mathcal{X} \in \mathcal{H}(\pi)$, $X \in \{0, 1, \ldots, n-1\}$, $Y \in \{\gamma \in \{0, 1, \ldots, n-1\} : X + \gamma \leq n\}$, $i \in \{0, 1, \ldots, n - X\}$, and $j \in \{\gamma \in \{0, 1, \ldots, n - Y\} : \gamma - i \geq X \ \vee \ i - \gamma \geq Y\}$. Then we define operator `2-string` as a relation `2-string` $: \mathcal{Y} \mapsto \mathcal{H}(\pi)$ which takes a string of vertices of size X that come after i-th vertex of $\mathcal{X}$ and a string of vertices of size Y that come after j-th vertex of $\mathcal{X}$ and interchanges them to create a new path $\mathcal{X}' \in \mathcal{H}(\pi)$. With this definition of `2-string`, we can define other operators as the latter with fixed X and Y to some values specific for each operator. For the fixed values of X and Y for all the operators except `2-opt` see Tab. 3.2.

The computational complexity of exploring the whole neighborhood $\mathcal{N}(\mathcal{X})$ for $\mathcal{N} \in \boldsymbol{\mathcal{N}}$ needs to be addressed as it is the core step of the improving procedure (see Alg. 3.13, line 6). Note, that all considered operators take two arguments (i

	`1-point`	`or-opt2`	`or-opt3`	`or-opt4`	`or-opt5`	`2-point`	`3-point`
$X =$	0	0	0	0	0	1	1
$Y =$	1	2	3	4	5	1	2

Table 3.2: All operators except `2-opt` can be defined as `2-string` with fixed values of X and Y.

and j) and all variations of the parameters need to be tried when exploring $\mathcal{N}(\mathcal{X})$. Therefore, the computational complexity of exploring the whole neighborhood must be $\mathcal{O}(n^{2+k})$. Here, k is the number of necessary cycles over the vertices to compute the improvement obtained by applying the operator on $\mathcal{X}$ with given parameters i and j. For the TSP, the improvement computation is straightforward without any cycle, therefore $k = 0$ and the whole neighborhood can be explored in $\mathcal{O}(n^2)$. Mladenović et al. [31] show, that the same holds for the TDP, if some additional structures are considered and a pre-processing step is performed. They derive the improvement for `2-opt` and some other operators. For `2-opt`, we use their result, slightly modified. Let $\mathcal{X} = (x_0, x_1, \ldots, x_i, x_{i+1}, \ldots, x_j, x_{j+1}, \ldots, x_n)$, and $\mathcal{X}' = (x_0, x_1, \ldots, x_i, x_j, \ldots, x_{i+1}, x_{j+1}, \ldots, x_n)$ be the original and `2-opt`ed path respectively. Then the obtained improvement $\Delta_{\texttt{2-opt}}(\mathcal{X}, i, j) = \mathit{cost}(\mathcal{X}') - \mathit{cost}(\mathcal{X})$ can be expressed as

$$\begin{aligned}\Delta_{\texttt{2-opt}}(\mathcal{X}, i, j) = {} & 2\mathcal{F}_i^{\mathcal{X}} - 2\mathcal{F}_n^{\mathcal{X}} + 2\mathcal{L}_{j+1}^{\mathcal{X}} + (\delta_i^{\mathcal{X}} + \delta_j^{\mathcal{X}} + d_{i|j}^{\mathcal{X}})(j-i) \\ & + (d_{i|j}^{\mathcal{X}} + d_{i+1|j+1}^{\mathcal{X}} - d_{i|i+1}^{\mathcal{X}} - d_{j|j+1}^{\mathcal{X}})(n-j),\end{aligned} \tag{3.5}$$

where $\delta_k^{\mathcal{X}}$ defined in Eq. (2.12),

$$\mathcal{F}_k^{\mathcal{X}} = \sum_{z=1}^{k} \delta_z^{\mathcal{X}}, \text{ and } \mathcal{L}_k^{\mathcal{X}} = \sum_{z=k}^{n} \delta_z^{\mathcal{X}} \tag{3.6}$$

are the pre-computed structures for $\mathcal{X}$ which are considered to be known at the time of the computation. Here and henceforward, we denote $d(x_i, x_j)$ as $d_{i|j}^{\mathcal{X}}$ to save horizontal space. For the proof, see [31]. A derivation of similar result for the `2-string` operator is shown next.

Assuming $0 < X$, $0 < Y$, $i + X < j$ and $j + Y < n$ we consider the original path to be $\mathcal{X} = (x_0, x_1, \ldots, x_i, x_{i+1}, \ldots, x_{i+X}, x_{i+X+1}, \ldots, x_j, x_{j+1}, \ldots, x_{j+Y}, x_{j+Y+1}, \ldots, x_n)$, and the operation $\mathcal{X}' = \texttt{2-string}(\mathcal{X}, X, Y, i, j)$ results in $\mathcal{X}' = (x_0, x_1, \ldots, x_i, x_{j+1}, \ldots, x_{j+Y}, x_{i+X+1}, \ldots, x_j, x_{i+1}, \ldots, x_{i+X}, x_{j+Y+1}, \ldots, x_n)$. For the graphical example of the operation see Fig 3.4. The cost of the original path is $\mathit{cost}(\mathcal{X}) = \delta_1^{\mathcal{X}} + \cdots + \delta_n^{\mathcal{X}}$ and the cost of the new path can be

computed as

$$
\begin{aligned}
cost(\mathcal{X}') = {} & \delta_1^{\mathcal{X}} + \cdots + \delta_i^{\mathcal{X}} + \\
& \delta_{i+1}^{\mathcal{X}} + \Delta_{-(i,i+1)}^{+(i,j+1)} + \cdots + \delta_{i+Y}^{\mathcal{X}} + \Delta_{-(i,i+1,\ldots,i+Y)}^{+(i,j+1,\ldots,j+Y)} + \\
& \delta_{i+X+1}^{\mathcal{X}} + \underbrace{\Delta_{-(i,i+1,\ldots,i+X,i+X+1)}^{+(i,j+1,\ldots,j+Y,i+X+1)}}_{=\Theta_1} + \cdots + \delta_j^{\mathcal{X}} + \Theta_1 + \\
& \delta_{j+1}^{\mathcal{X}} + \Theta_1 + \Delta_{-(j,j+1)}^{+(j,i+1)} + \cdots + \delta_{i+X}^{\mathcal{X}} + \Theta_1 + \Delta_{-(j,j+1,\ldots,j+X)}^{+(j,i+1,\ldots,i+X)} + \\
& \delta_{j+Y+1}^{\mathcal{X}} + \underbrace{\Theta_1 + \Delta_{-(j,j+1,\ldots,j+Y,j+Y+1)}^{+(j,i+1,\ldots,i+X,j+Y+1)}}_{=\Theta_2} + \cdots + \delta_n^{\mathcal{X}} + \Theta_2,
\end{aligned}
\tag{3.7}
$$

where we use the following notation: $\Delta_{-(c,\ldots,d)}^{+(a,\ldots,b)} = d_{(a,\ldots,b)}^{\mathcal{X}} - d_{(c,\ldots,d)}^{\mathcal{X}}$, and $d_{(a,b,c,\ldots,d,e)}^{\mathcal{X}} = d_{a|b}^{\mathcal{X}} + d_{b|c}^{\mathcal{X}} + \cdots + d_{d|e}^{\mathcal{X}}$. The improvement obtained by applying `2-string` on path $\mathcal{X}$ is given by

$$
\begin{aligned}
\Delta_{\texttt{2-string}}^{X,Y}(\mathcal{X},\, i,\, j) = {} & cost(\mathcal{X}') - cost(\mathcal{X}) = \\
& \delta_{i+1}^{\mathcal{X}} - \delta_{j+1}^{\mathcal{X}} + \Delta_{-(i,i+1)}^{+(i,j+1)} + \\
& \qquad \cdots + \delta_{i+Y}^{\mathcal{X}} - \delta_{j+Y}^{\mathcal{X}} + \Delta_{-(i,i+1,\ldots,i+Y)}^{+(i,j+1,\ldots,j+Y)} + \\
& \delta_{j+1}^{\mathcal{X}} - \delta_{i+1}^{\mathcal{X}} + \Theta_1 + \Delta_{-(j,j+1)}^{+(j,i+1)} + \\
& \qquad \cdots + \delta_{j+X}^{\mathcal{X}} - \delta_{i+X}^{\mathcal{X}} + \Theta_1 + \Delta_{-(j,j+1,\ldots,i+X)}^{+(j,i+1,\ldots,i+X)} + \\
& \Theta_1(j - i - X) + \Theta_2(n - j - Y).
\end{aligned}
\tag{3.8}
$$

With the use of identities

$$
\begin{aligned}
\Delta_{-(i,i+1)}^{+(i,j+1)} &= \delta_i^{\mathcal{X}} - \delta_{i+1}^{\mathcal{X}} + d_{i|j+1}^{\mathcal{X}}, \\
\Delta_{-(i,i+1,\ldots,i+Y)}^{+(i,j+1,\ldots,j+Y)} &= \delta_i^{\mathcal{X}} - \delta_{i+Y}^{\mathcal{X}} + d_{i|j+1}^{\mathcal{X}} + \delta_{j+Y}^{\mathcal{X}} - \delta_{j+1}^{\mathcal{X}}, \\
\Delta_{-(j,j+1)}^{+(j,i+1)} &= \delta_j^{\mathcal{X}} - \delta_{j+1}^{\mathcal{X}} + d_{j|i+1}^{\mathcal{X}}, \text{ and} \\
\Delta_{-(j,j+1,\ldots,j+X)}^{+(j,i+1,\ldots,i+X)} &= \delta_j^{\mathcal{X}} - \delta_{j+X}^{\mathcal{X}} + d_{j|i+1}^{\mathcal{X}} + \delta_{i+X}^{\mathcal{X}} - \delta_{i+1}^{\mathcal{X}},
\end{aligned}
\tag{3.9}
$$

we can rewrite Eq. (3.8) as

$$
\begin{aligned}
& \Delta_{\texttt{2-string}}^{X,Y}(\mathcal{X},\, i,\, j) = \\
& \quad \delta_i^{\mathcal{X}} - \delta_{j+1}^{\mathcal{X}} + d_{i|j+1}^{\mathcal{X}} + \cdots + \delta_i^{\mathcal{X}} - \delta_{j+1}^{\mathcal{X}} + d_{i|j+1}^{\mathcal{X}} + \\
& \quad \delta_j^{\mathcal{X}} - \delta_{i+1}^{\mathcal{X}} + \Theta_1 + d_{j|i+1}^{\mathcal{X}} + \cdots + \delta_j^{\mathcal{X}} - \delta_{i+1}^{\mathcal{X}} + \Theta_1 + d_{j|i+1}^{\mathcal{X}} + \\
& \quad \Theta_1(j - i - X) + \Theta_2(n - j - Y),
\end{aligned}
\tag{3.10}
$$

which can be further simplified into the following form:

$$
\begin{aligned}
& \Delta_{\texttt{2-string}}^{X,Y}(\mathcal{X},\, i,\, j) = \\
& \quad \Lambda_1 Y + \Lambda_2 \left(j - i - X\right) + \Lambda_3 X + \Lambda_4 \left(n - j - Y\right),
\end{aligned}
\tag{3.11}
$$

where

$$\Lambda_1 = \delta^{\mathcal{X}}_i - \delta^{\mathcal{X}}_{j+1} + d^{\mathcal{X}}_{i|j+1}, \tag{3.12}$$
$$\Lambda_2 = \Lambda_1 + \delta^{\mathcal{X}}_{j+Y} - \delta^{\mathcal{X}}_{i+X+1} + d^{\mathcal{X}}_{j+Y|i+X+1}, \tag{3.13}$$
$$\Lambda_3 = \Lambda_2 + \delta^{\mathcal{X}}_{j} - \delta^{\mathcal{X}}_{i+1} + d^{\mathcal{X}}_{j|i+1}, \tag{3.14}$$
$$\Lambda_4 = \Lambda_3 + \delta^{\mathcal{X}}_{i+X} - \delta^{\mathcal{X}}_{j+Y+1} + d^{\mathcal{X}}_{i+X|j+Y+1}. \tag{3.15}$$

Note that with different assumptions the computation of $\Delta^{X,Y}_{\texttt{2-string}}(\mathcal{X},\, i,\, j)$ may differ, but its derivation will be analogous to the one we have shown. Let us further state two observations that we encountered when deriving the improvement for all the special cases and which may bring major simplification to one who may follow our path. The two observations are:

$$X = Y \Rightarrow \Delta^{X,Y}_{\texttt{2-string}}(\mathcal{X},\, i,\, j) = \Delta^{X,Y}_{\texttt{2-string}}(\mathcal{X},\, j,\, i), \tag{3.16}$$
$$j < i \Rightarrow \Delta^{X,Y}_{\texttt{2-string}}(\mathcal{X},\, i,\, j) = \Delta^{Y,X}_{\texttt{2-string}}(\mathcal{X},\, j,\, i). \tag{3.17}$$

3.3.8 Proposed: Ms-GVNS

The best metaheuristic design is done in two phases. The first phase chooses several promising sets of neighborhoods considered in the improvement procedure. In the second phase, a number of methods' configurations are tested, and the best is selected as the proposed method. We describe the process in extensive detail in Appx. A. All tested methods, including the reference GILS-RVND [30], are implemented in `C++` under the same framework. All possible parts of code are shared between different configurations in order to ensure fairness. This include improvement calculations of all operators as they are described in Sec. 3.3.7, despite the fact, that the authors of [30] use different improvement calculations for their GILS-RVND. The authors were so kind to provide us with their code so we could ensure that our implementation of GILS-RVND is not worse, in any sense, than their implementation.

We call the best overall metaheuristic *multi-start GVNS* (Ms-GVNS). Its full specification follows. It is based on the G+G general scheme (1c) but uses deterministic greedy construction (3a), hence the name. Its parameters are the following: $\boldsymbol{\mathcal{N}} = (\mathcal{N}_{\texttt{2-opt}}, \mathcal{N}_{\texttt{1-point}}, \mathcal{N}_{\texttt{or-opt2}}, \mathcal{N}_{\texttt{or-opt3}}, \mathcal{N}_{\texttt{or-opt4}})$, $\mathcal{P} = (4, 8, 12)$, and $j_{max} = \lceil size(i)/5 \rceil$, where $size(i)$ is the size of the solved instance i. Furthermore, it uses VND (the deterministic one) (6a) and leaves the inner loop after j_{max} iterations without improvement (4b).

3.4 TDP extensions

Ways to model the *mobile search* better than by plain TDP are discussed in this section. We introduce modifications of the TDP that consider an extended model of the robot's kinematics and differing gains of cover-locations in Subsec. 3.4.1. Additional extension that acknowledges sensing on the way between cover-locations is developed in Subsec. 3.4.1. The last improvement in Subsec. 3.4.3 suggests replanning during the search as a way to cope with changing cover-locations' gains.

We use the same metaheuristic as developed for the TDP in Sec. 1.3.3, i.e., the Ms-GVNS, to solve the modified problems. The algorithm and its parameters stay identical and the improvement computations as in Sec. 3.3.7 are updated to suit each individual variant.

3.4.1 ATDP, GSP, AGSP

We extend the TDP defined in Sec. 2.2 by considering two additional traits of the graph $G = (V, E)$:

- $\alpha : V \times V \times V \to \mathbb{R}_0^+$: a non-negative cost $\alpha(i, j, k)$ associated with each triplet (v_i, v_j, v_k), of the graph's vertices or alternatively with each pair $(e_{i,j}, e_{j,k})$ of consecutive edges that share one common endpoint v_j. The costs are assumed symmetrical, i.e., $\alpha(i, j, k) = \alpha(k, j, i)$.

- $w : V \to \mathbb{R}_0^+$: a non-negative weight $w(i)$ associated with each vertex v_i.

We denote the problem defined by tuple

- (G, d, α, s) as **ATDP**,

- (G, d, w, s) as **GSP**, and

- (G, d, α, w, s) as **AGSP**.

Then, we define the total AGSP cost of a Hamiltonian path $\mathcal{X} = (x_0 = s, x_1, \ldots, x_n)$ in graph G as

$$\begin{aligned} cost(\mathcal{X}) &= \sum_{k=1}^{n} w(k) \cdot \sum_{i=1}^{k} \left(d(x_{i-1}, x_i) + \alpha(x_{i-2}, x_{i-1}, x_i)\right) \\ &= \sum_{k=1}^{n} w(k) \cdot \delta_k^{\mathcal{X}}, \end{aligned} \tag{3.18}$$

where $\alpha(x_{-1}, x_0, x_1)$, that would formally be undefined, can be zero or $\alpha(x_{-1}, x_0, x_1) = \alpha_0(x_1)$, where $\alpha_0 : V \to \mathbb{R}_0^+$ is an additional one-dimensional cost provided for all vertices. Notice that the meaning of $\delta_k^{\mathcal{X}}$ in Eq. (3.18) differs from the TDP, Eq. (2.12). Let us note that the improvement computations as in Sec. 3.3.7 for the new extended problems are different than for the TDP and must be derived anew analogously as for the TDP.

Since AGSP is the most general problem, the total costs for ATDP and GSP can be defined in terms of Eq. 3.18, where $\forall\, v_i, v_j, v_k \in V : \alpha(v_i, v_j, v_k) = 0$ and $\forall\, v_i \in V : w(i) = 1$, respectively. Unlike ATDP and AGSP, the GSP is not an original problem, i.e., it is already known in the literature. The GSP stands for *graph search problem* that was originally introduced in Koutsoupias et al. [18], and its first applications were built around the area of web searching. The connection between TSP, TDP, and GSP is also discussed in [21]. Furthermore, the GSP was utilized for solving the *mobile search* in [10] and in our previous work [33]. Interpretations of costs $\alpha(i, j, k)$ and weights $w(i)$ with respect to the *mobile search* are discussed next.

Mobile search formulations that consider the **three dimensional costs** $\alpha(i, j, k)$ can better model the robot's kinematics. To achieve the better model, the original cost $d(i, j)$ does not represent the length of the shortest path from location l_i to l_j but instead the shortest time of travel from l_i to l_j. Similarly, the new cost $\alpha(i, j, k)$ represents the shortest time the robot needs to spend (e.g., by turning) in location l_j while assuming it reached l_j from l_i, and its next goal is l_k. In our idealized case, paths through the environment are represented as poly-lines, and the robot can follow given path by a series of simple maneuvers consisting of going straight with linear velocity v_{lin} (m/s), and turning on the spot with angular velocity v_{ang} (rad/s). Therefore, the time $d(i, j)$ of travel from l_i to l_j is equal to

$$d(i, j) = \frac{\rho_{i,j}}{v_{lin}} + \frac{\phi_{i,j}}{v_{ang}}, \tag{3.19}$$

where $\rho_{i,j}$ is the length of the shortest poly-line from l_i to l_j, and $\phi_{i,j}$ is the robot's turning angle accumulated over the same poly-line. The accumulated turning angle $\phi_{i,j}$ is computed as the sum of absolute values of all the turning angles on the way from l_i to l_j. The time $\alpha(i, j, k)$ spent on location l_j,

$$\alpha(i, j, k) = \frac{\beta_{i,j,k}}{v_{ang}}, \tag{3.20}$$

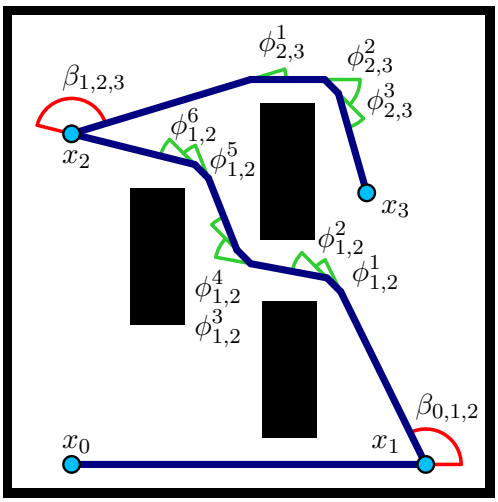
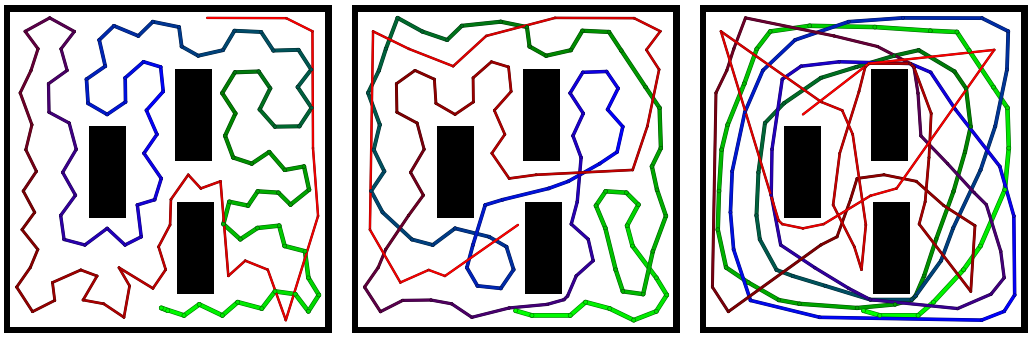


Figure 3.5: Exaple of a simple route $\mathcal{X} = (x_0, x_1, x_2, x_3)$ inside a polygonal environment, where all non-zero turning angles of the robot are marked. The angles dependent on three locations, i.e., $\beta_{0,1,2}$, $\beta_{1,2,3}$, are shown in red. The accumulated angles dependent only on two locations, i.e., $\phi_{1,2} = \sum_i |\phi^i_{1,2}|$, $\phi_{2,3} = \sum_i |\phi^i_{2,3}|$, are shown in green.



(a) : $v_{lin} \ll v_{ang}$ **(b) :** $v_{lin} \approx v_{ang}$ **(c) :** $v_{lin} \gg v_{ang}$

Figure 3.6: The effect of velocity constants v_{lin}, v_{ang} on an optimized ATDP route. If $v_{lin} \ll v_{ang}$, then the optimization prefers short distances. On the other hand, if $v_{lin} \gg v_{ang}$, then small turning angles are preferred. The two features are, to some extend, balanced if $v_{lin} \approx v_{ang}$.

is proportional to the turning angle $\beta_{i,j,k}$ on location l_j, which is dependent on the previous location l_i and the current goal l_k. An example of a simple route consisting of several locations to visit and the corresponding turning angles of two types (ϕ and β) are shown in Fig. 3.5. Let us note, that the choice of velocity constants v_{lin}, v_{ang} significantly impacts the character of the resulting route after it is optimized. This effect in case of the ATDP is shown in Fig. 3.6.

Mobile search formulations that consider the **weights** $w(i)$ are better at estimating the probability of finding the searched object when visiting location l_i and, therefore, better guide the search towards promising areas. For example, at the beginning of the search, visiting an open area location should be preferred over visiting a location inside of a tiny room, assuming uniform probability distribution for the searched object's location, as we do.

The weight $w(i)$ associated with location l_i could be determined in various creative ways; however, none of them would be quite accurate as we assume static weights in our formulations, but in reality, they change as the search progresses. We define the weight of location l_i in the most straightforward way as an area of a *region sensed from* the location, i.e.,

$$w(i) = \mathit{Area}(\mathcal{V}(l_i)), \tag{3.21}$$

where $\mathcal{V}(l_i)$ is defined by Eq. (2.11). We deal with the non-static character of weights in Sec. 3.4.3.

3.4.2 GSP2, AGSP2

We define two more variants of the TDP similar to the GSP and AGSP, but this time with two-dimensional weights. Let us consider one more trait of the graph $G = (V, E)$:

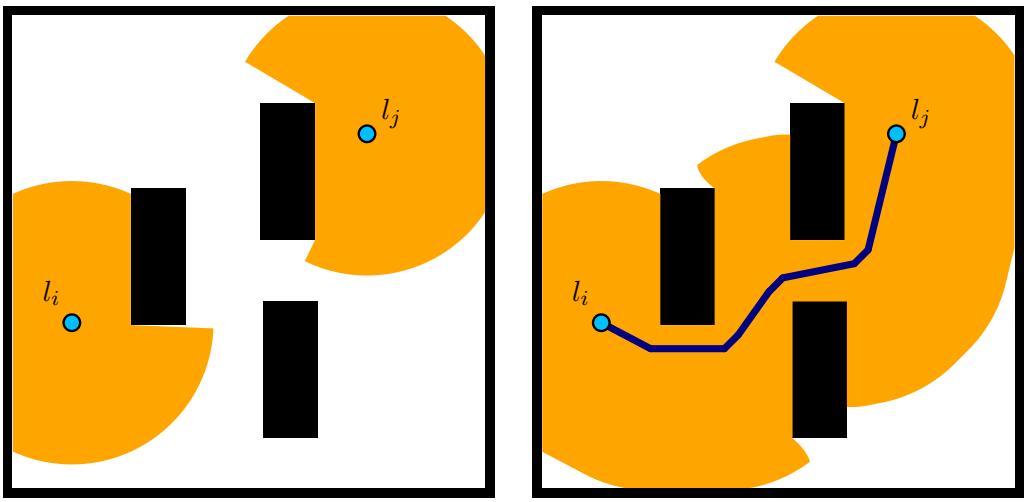
- $w_2 : E \to \mathbb{R}_0^+$: a non-negative weight $w_2(i, j)$ associated with each edge $e_{i,j} \in E$. The weights are symmetrical, i.e., $w_2(i, j) = w_2(j, i)$.

We denote the problem defined by tuple

- (G, d, w_2, s) as **GSP2**, and
- (G, d, α, w_2, s) as **AGSP2**.

Then, we define the total AGSP2 cost of a Hamiltonian path $\mathcal{X} = (x_0 = s, x_1, \ldots, x_n)$ in graph G as

$$\begin{aligned} \mathit{cost}(\mathcal{X}) &= \sum_{k=1}^{n} w_2(k-1, k) \cdot \sum_{i=1}^{k} d(x_{i-1}, x_i) + \alpha(x_{i-2}, x_{i-1}, x_i) \\ &= \sum_{k=1}^{n} w_2(k-1, k) \cdot \delta_k^{\mathcal{X}}. \end{aligned} \tag{3.22}$$



(a) : $\mathcal{V}(l_i)$, $\mathcal{V}(l_j)$ **(b) :** $\mathcal{V}(l_i, l_j)$

Figure 3.7: Comparison of weights' computation for GSP, AGSP (left) and GSP2, AGSP2 (right).

Since AGSP2 is more general than GSP2, the cost for GSP2 can be defined in terms of Eq. 3.22, where $\forall\, v_i, v_j, v_k \in V : \alpha(v_i, v_j, v_k) = 0$. Again, let us note that the improvements for the introduced problems must be derived anew analogously as for the TDP.

Mobile search formulations that consider the **two-dimensional weights** $w_2(i, j)$ allow sensing during the whole search, i.e., in between the locations to visit and not only when the robot reaches one of these locations. This, again, better models the reality of our problem defined in Sec. 2.1.

We define the weight of two locations l_i, l_j in the most straightforward way, analogously as the one-dimensional weight for the GSP and AGSP, as

$$w(i, j) = \mathit{Area}(\mathcal{V}(l_i, l_j)), \tag{3.23}$$

where $\mathcal{V}(l_i, l_j)$ is the whole *region sensed* while following the shortest path from l_i to l_j. The comparison between $\mathcal{V}(l_i, l_j)$ and the classical one-location $\mathcal{V}(l_i)$ is shown in Fig. 3.7. In practice, we acquire the region $\mathcal{V}(l_i, l_j)$ by finely discretizing the path, computing all *sub-regions sensed* from the discretized points of the path, and finally computing the union of all these sub-regions.

3.4.3 Replanning

The final improvement that we suggest is replanning during the search. Replanning is, originally, an online process of creating a new plan during the execution of the current plan. However, if we consider that we have available an implementation of the whole search scheme, Alg. 3.2 where a simple simulation replaces the actual robot's navigation, we can create an offline plan that is based on replanning even before starting the search execution. The refined plan is returned by Alg. 3.2 as the truly visited sequence of goals $\boldsymbol{l}^{true}$. For our experiments, we choose this scenario. Nevertheless, in a real application to shorten the time of planning, i.e., waiting before starting the search, the replanning might be performed online during the search. We expect the online and offline approaches to be equivalent if the considered simulation sufficiently well describes the robot's kinematics.

In all cases, the new plan's creation works as follows: already visited locations are discarded from the new plan, weights of the rest are recomputed while considering the so-far covered region $\mathcal{W}_{cov}$, and the resulting new instance of the *routing problem* is solved. Replanning is realized by the `RePlan` procedure in the general search scheme, Alg. 3.2. It is only meaningful for formulations that consider the weights of locations or paths, i.e., GSP, AGSP, GSP2, and AGSP2. During replanning, the new weights for GSP and AGSP are computed as

$$w(i) = Area(\mathcal{V}(l_i) \setminus \mathcal{W}_{cov}), \tag{3.24}$$

and for GSP2 and AGSP2 as

$$w(i,j) = Area(\mathcal{V}(l_i, l_j) \setminus \mathcal{W}_{cov}). \tag{3.25}$$

If the replanning is used in Alg. 3.2, we denote the corresponding planning algorithms as GSP-RP, AGSP-RP, GSP2-RP, and AGSP2-RP, respectively. Finally, one last rule is included within those planning variants. If $\mathcal{V}(l_i) \setminus \mathcal{W}_{cov} = \emptyset$ for some unvisited location l_i, then this location is entirely discarded, i.e., not included in the next and all later plans.

Chapter 4

Computational evaluation

4.1 TDP: Ms-GVNS vs. GILS-RVND

First, we demonstrate the properties of the developed metaheuristic for the TDP on its own in this section; then, we compare complex solutions to the *mobile search* where the metaheuristic is integrated (solving various problems derived from the TDP) in the next Sec. 4.2.

Thorough computational evaluation of Ms-GVNS and its comparison with the state-of-the-art method for the TDP — GILS-RVND [30] — follows. For the final evaluation, we use a different implementation of the algorithms from the one employed for designing the proposed Ms-GVNS and tuning its parameters in Appx. A. While the original heavy implementation was optimized for the most effortless use with much variability, the ultimate evaluation is realized under a new, more straightforward implementation, aiming towards the best possible performance. Both methods are again implemented in `C++`, sharing all possible pieces of code. Ms-GVNS (*the proposed*) is parametrized as in Sec. A.4, and GILS-RVND (*the reference*) as in [30]. All experiments described in this section are executed on a personal computer with Intel® Core™ i7-7700 CPU (3.60 GHz), 32 GB of RAM, and Ubuntu 18.04.1 LTS. The implementation is single-threaded, and only one physical core of the CPU is used for each experiment.

The proposed is tested against *the reference* on several sets of standard benchmark instances generated by Salehipour et al. [29]. The available sets consider 10, 20, 50, 100, 200, 500, and 1000 customers, respectively, and each is composed of 20 random instances. For the evaluation, we use three different approaches that we call *time-limits*, *TTT-plots*, and *fixed-iters*.

1. ***Time-limits:*** our key results — only a computational time limit is given to the method, 50 runs are performed, and the resulting solution costs are recorded. This approach is realized on instances of sizes 200, 500, and 1000, and the considered time limits are 1, 2, 5, 10, 20, 50, and 100 seconds.

2. ***TTT-plots:*** plots from 200 executions are obtained (the same way as in Appx. A), and the probability $p_{pr} \approx P(RT_p < RT_r)$ is computed as described in Sec. 1.3.5. Here, RT_p and RT_r are random variables representing the time needed by *the proposed* and *the reference* respectively to find a solution that is as good as given c_{goal}. This approach is realized on instances of sizes 10, 20, 50, 100, 200, and 500, and the given target value c_{goal} is the optimum for smaller instances (sizes 10, 20, 50) where it is known [29, 30], and the best solution reported by [30] worsened by 1% for the rest of instances.

3. ***Fixed-iters:*** this type of evaluation is the most common in the literature. The algorithm is given a fixed number of iterations ($i_{max} = 10$), and 10 runs are performed. The returned solution costs and the computational times are recorded. This approach is realized on instances of sizes 10, 20, 50, 100, 200, and 500.

In the tables presented hereafter, c_{best} is the best-known solution, Best and Mean denote the best and the mean solution cost found by the considered algorithm respectively, %bG, and %mG are the best, and a mean percentage gap from c_{best} computed as $100 \cdot (\text{Best} - c_{best})/c_{best}$, and $100 \cdot (\text{Mean} - c_{best})/c_{best}$ respectively, and Time is the average computing time over 10 executions. For instances with up to 200 customers, c_{best} corresponds to values reported by [30], while for 500 and 1000-customer instances they are the minimum of values reported by [30], [73], and [74]. In the context of *TTT-plots* results, we report TTT, as the average (over 200 runs) time to target solution, and $\%p_{pr}$ as the probability $p_{pr} \approx P(RT_p < RT_r)$ computed from TTT plots of both methods, in percents. An over-lined symbol (e.g., $\overline{\%\text{bG}}$) indicates that the value is averaged over instances of the same size. In some tables, we use blue and orange colors to emphasize %bG, %mG, Time, TTT, and $\%p_{pr}$ values where Ms-GVNS and GILS-RVND, respectively, performed better than the other. If the methods performed equally, then none of the values is shown in color. If one of the values is in color, but the values numerically appear equal, then the difference is non-zero and was lost after rounding. Additionally, we note that within generating *fixed-iters* results on instances of sizes 10-500, we performed the same experiments as Silva et al. [30], however with our own implementation, improvement calculations, different random number generator seeds, and using more powerful hardware. For %bG and %mG, our obtained results are identical (except some small statistical error), to the ones reported by the authors. On the other hand, our values of Time are significantly lower. Thus, we report unchanged values of %bG, %mG from [30], and our updated values of Time in our results.

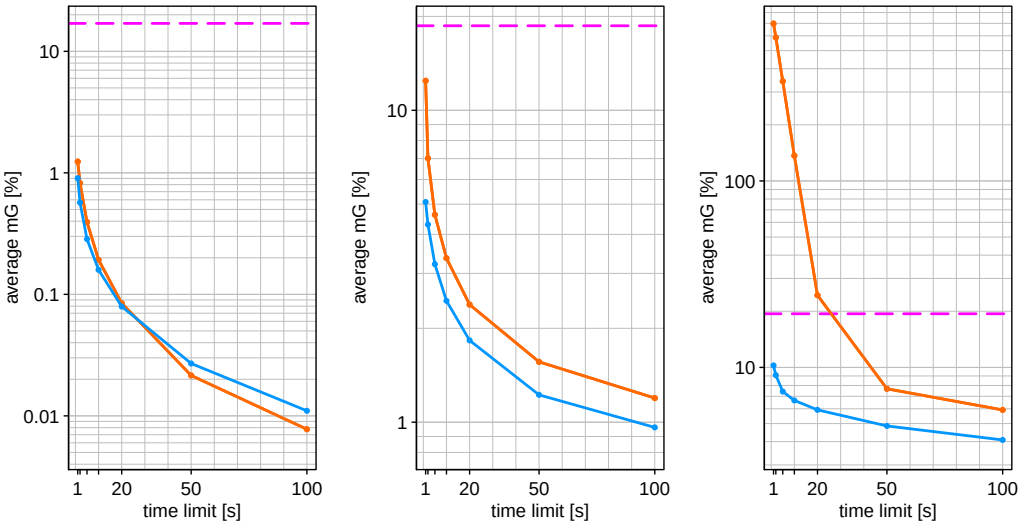


Figure 4.1: Convergence of the average mean gap from Tables 4.1, 4.2, and 4.3 for Ms-GVNS (blue) and GILS-RVND (orange). The dashed magenta horizontal line is the average %mG of the upper bound.

To produce our main results, *time-limits*, we simulate a scenario in which the algorithms are allowed to run only for a fixed time t_{max}, after which they immediately return a solution, and its cost is recorded. Seven different values for t_{max} are used (1, 2, 5, 10, 20, 50, and 100 seconds), and 50 runs are executed for each experimental setup consisting of a method, an instance, and a time-limit. The values of %mG over 50 runs on instances with 200, 500, and 1000 customers are presented in Tables 4.1, 4.2, and 4.3, respectively. The one extra column, %mG of UB, shows the mean gaps of upper bounds from the best-known solutions. The upper bounds are obtained by a purely greedy algorithm used to generate the initial solution in Ms-GVNS. For 200-customer instances, both Ms-GVNS and GILS-RVND return a better solution than the upper bound even after one second of computing time. As expected, the proposed method converges more quickly towards c_{best}. After 100 seconds, the solution returned by both methods is of similar quality. For 500-customer instances, the values after one second are closer to the upper bound, and after 100 seconds, Ms-GVNS typically still returns higher-quality solutions.

The convergence of the average %mG can be better tracked in Fig. 4.1, which shows the last rows (avg) of these tables plotted as graphs. See the graph for the largest instances with 1000 customers. Here, after one second, *the proposed* returns a solution better than the upper bound and converges to the best-known solution. However, *the reference* starts significantly above the upper bound and gets below it typically after no less than 20 seconds. This behavior is caused by generating the initial solution in a randomized way as it is done by GRASP. Such an initial solution can be very far from the optimum, and it takes many operations

Inst	%mG														
	UB	GILS-RVND							Ms-GVNS						
		t_{max} : 1	2	5	10	20	50	100	1	2	5	10	20	50	100
R1	18.31	1.03	0.65	0.31	0.16	0.08	0.03	0.00	0.83	0.60	0.24	0.12	0.05	0.01	0.01
R2	13.15	1.24	1.02	0.59	0.35	0.24	0.07	0.05	0.93	0.71	0.48	0.31	0.20	0.11	0.06
R3	20.61	1.08	0.54	0.29	0.15	0.08	0.02	0.01	0.69	0.35	0.13	0.10	0.06	0.03	0.01
R4	12.65	1.12	0.74	0.30	0.08	0.04	0.01	0.00	0.71	0.41	0.08	0.03	0.01	0.00	0.00
R5	14.85	1.26	0.87	0.46	0.27	0.18	0.07	0.02	0.81	0.53	0.34	0.30	0.15	0.07	0.04
R6	13.28	1.46	0.94	0.41	0.13	0.05	0.00	0.00	1.33	0.70	0.23	0.11	0.03	0.00	0.00
R7	18.39	1.49	1.17	0.53	0.23	0.08	0.01	0.00	1.17	0.70	0.42	0.19	0.09	0.03	0.00
R8	16.60	1.15	0.85	0.29	0.12	0.01	0.00	0.00	0.90	0.54	0.22	0.13	0.05	0.01	0.00
R9	16.94	1.17	0.74	0.46	0.25	0.06	0.00	0.00	1.03	0.78	0.39	0.30	0.09	0.00	0.00
R10	18.03	1.03	0.99	0.44	0.17	0.06	0.03	0.00	1.09	0.70	0.21	0.10	0.05	0.00	0.00
R11	11.49	0.81	0.50	0.24	0.12	0.08	0.02	0.01	0.38	0.29	0.13	0.06	0.03	0.01	0.00
R12	17.78	1.06	0.61	0.34	0.15	0.09	0.03	0.01	0.61	0.31	0.19	0.09	0.05	0.02	0.00
R13	16.66	1.13	0.79	0.35	0.23	0.06	0.02	0.00	0.43	0.23	0.07	0.03	0.01	0.00	0.00
R14	17.04	1.52	0.81	0.43	0.16	0.08	0.01	0.00	1.12	0.54	0.32	0.15	0.05	0.01	0.00
R15	12.69	1.17	0.83	0.40	0.13	0.05	0.00	0.00	0.74	0.51	0.24	0.12	0.04	0.00	0.00
R16	16.96	1.84	1.04	0.68	0.30	0.14	0.02	0.00	1.55	1.05	0.67	0.37	0.19	0.06	0.03
R17	20.35	0.86	0.61	0.32	0.17	0.06	0.02	0.01	0.79	0.61	0.41	0.21	0.11	0.03	0.01
R18	15.52	1.77	1.02	0.33	0.22	0.09	0.02	0.00	1.24	0.69	0.39	0.26	0.22	0.11	0.07
R19	13.22	1.31	0.95	0.45	0.34	0.16	0.05	0.03	0.97	0.71	0.43	0.18	0.12	0.04	0.00
R20	35.21	1.26	0.77	0.25	0.11	0.01	0.00	0.00	0.77	0.43	0.11	0.01	0.00	0.00	0.00
avg	**16.99**	**1.24**	**0.82**	**0.39**	**0.19**	**0.08**	**0.02**	**0.01**	**0.90**	**0.57**	**0.28**	**0.16**	**0.08**	**0.03**	**0.01**

Table 4.1: *Time-limits* results on instances with 200 customers. All times are in seconds.

Inst	%mG														
	UB	GILS-RVND							Ms-GVNS						
		t_{max} : 1	2	5	10	20	50	100	1	2	5	10	20	50	100
R1	19.1	12.1	6.8	4.8	3.7	2.6	1.6	1.5	6.0	5.2	4.0	3.1	2.2	1.7	1.2
R2	19.9	12.8	7.3	4.5	3.4	2.1	1.4	1.0	5.6	4.5	3.4	2.2	1.5	1.0	0.7
R3	23.8	11.7	6.8	4.8	3.3	2.4	1.5	1.1	5.5	4.5	3.1	2.4	1.7	1.2	0.9
R4	15.9	12.2	7.1	4.8	3.3	2.5	1.7	1.2	5.1	4.2	3.4	2.7	2.2	1.5	1.2
R5	21.7	11.6	6.8	4.7	3.3	2.3	1.7	1.2	5.8	4.7	3.5	2.7	2.2	1.3	1.1
R6	19.9	13.3	6.9	4.4	3.1	2.2	1.5	1.0	5.2	4.2	2.9	2.3	1.5	1.1	0.9
R7	19.9	11.7	7.4	4.8	3.7	2.8	1.7	1.5	5.3	4.7	3.6	2.9	2.2	1.5	1.3
R8	17.2	13.1	7.0	4.8	3.4	2.4	1.6	1.3	5.5	4.9	3.5	2.7	1.9	1.3	0.9
R9	23.8	12.5	7.2	4.5	3.4	2.3	1.5	1.1	4.0	3.5	2.8	2.4	1.8	1.3	1.1
R10	22.8	13.4	7.1	4.9	3.3	2.5	1.5	1.1	3.9	3.1	2.6	2.1	1.6	1.2	0.9
R11	27.3	11.8	6.2	3.8	2.8	1.9	1.2	0.9	4.4	3.6	2.6	1.8	1.4	0.9	0.7
R12	17.3	13.7	7.6	4.9	3.6	2.5	1.6	1.2	5.5	4.9	3.9	2.9	2.3	1.4	1.1
R13	11.6	12.0	6.7	4.7	3.5	2.3	1.7	1.3	4.8	4.0	2.8	2.2	1.6	1.1	0.8
R14	14.2	12.5	7.3	4.5	3.2	2.5	1.6	1.3	4.6	4.1	3.0	2.1	1.8	1.1	1.0
R15	14.1	12.0	7.4	5.0	3.4	2.6	1.7	1.3	3.9	3.2	2.6	1.8	1.3	0.9	0.8
R16	18.7	12.2	6.7	4.3	3.2	2.5	1.6	1.1	5.3	4.2	3.1	2.4	1.8	1.0	0.8
R17	16.4	12.0	6.6	4.3	3.3	2.4	1.6	1.3	5.0	4.6	3.2	2.6	1.7	1.2	0.8
R18	18.3	11.8	6.6	4.6	3.2	2.1	1.3	1.0	5.7	4.8	3.6	2.6	1.9	1.2	0.9
R19	13.1	12.7	7.5	4.7	3.6	2.3	1.4	1.1	5.4	4.6	3.2	2.5	1.7	1.1	0.7
R20	18.4	13.7	7.2	4.7	3.3	2.3	1.6	1.2	5.2	4.5	3.4	2.7	2.2	1.4	1.2
avg	**18.7**	**12.4**	**7.0**	**4.6**	**3.4**	**2.4**	**1.6**	**1.2**	**5.1**	**4.3**	**3.2**	**2.5**	**1.8**	**1.2**	**1.0**

Table 4.2: *Time-limits* results on instances with 500 customers. All times are in seconds.

Inst	%mG													
	UB	GILS-RVND							Ms-GVNS					
		t_{max} : 1	2	5	10	20	50	100	1	2	5	10	20	50	100
R1	18.3	676.1	502.6	247.2	93.6	12.3	6.9	5.7	9.4	8.1	6.1	5.6	5.2	4.2	3.5
R2	23.6	679.5	626.6	350.8	132.2	21.2	7.7	5.9	9.8	8.4	6.8	6.2	5.3	4.4	3.8
R3	15.6	685.2	588.8	355.2	117.5	17.9	6.9	5.4	10.7	9.9	7.8	6.6	5.7	4.6	3.8
R4	15.7	673.8	664.0	328.9	133.6	21.0	7.5	5.6	8.7	7.3	6.2	5.4	4.9	4.3	3.7
R5	21.8	766.3	586.0	313.7	142.4	21.5	7.5	5.6	9.3	8.1	6.3	5.8	5.4	4.7	4.2
R6	19.0	629.9	586.6	342.0	136.5	23.9	7.9	5.8	10.9	9.9	7.9	7.0	6.4	5.3	4.6
R7	19.9	720.8	636.7	368.9	140.2	27.3	7.9	6.5	10.9	9.6	7.8	7.3	6.6	5.7	4.9
R8	18.6	760.0	542.4	348.5	126.4	23.4	7.3	5.5	10.5	9.3	7.8	7.0	6.1	4.8	4.0
R9	22.6	675.8	617.0	368.8	126.8	24.5	8.3	6.2	11.7	10.9	8.9	8.1	7.0	5.6	4.5
R10	15.6	761.7	508.9	312.0	160.7	24.9	8.0	6.1	11.1	10.0	8.2	7.6	6.5	5.5	4.6
R11	20.6	688.0	586.8	335.0	137.0	25.9	7.7	6.2	11.5	9.9	7.4	7.0	6.1	4.8	4.0
R12	19.8	687.7	584.5	381.2	153.9	27.2	8.3	6.5	10.5	9.5	7.9	7.0	6.4	5.3	4.4
R13	19.1	656.3	599.2	355.8	143.3	26.4	7.6	6.0	11.3	10.1	8.1	6.6	6.0	4.9	4.1
R14	17.3	683.0	556.5	365.2	145.0	24.8	7.2	5.7	8.4	7.2	6.2	5.4	4.9	4.0	3.5
R15	15.9	674.2	645.3	327.8	143.7	23.6	7.7	5.7	9.2	8.1	6.3	5.7	4.9	4.2	3.5
R16	17.5	692.5	562.0	358.4	138.1	27.8	7.8	5.9	11.3	10.0	8.3	7.3	6.1	4.7	3.9
R17	20.3	661.2	571.6	316.9	144.1	27.0	7.8	5.9	10.2	9.4	8.5	7.4	6.6	5.3	4.4
R18	23.8	670.4	649.2	365.6	134.8	29.0	8.3	6.4	10.8	9.7	7.8	7.1	6.5	5.5	4.5
R19	18.7	737.2	593.3	375.0	141.9	28.7	7.7	5.8	9.5	8.1	6.9	6.3	5.7	4.6	3.8
R20	24.1	776.6	559.6	333.2	142.1	29.1	7.9	6.0	9.2	8.3	7.2	6.6	6.0	4.8	4.1
avg	**19.4**	**697.8**	**588.4**	**342.5**	**136.7**	**24.4**	**7.7**	**5.9**	**10.2**	**9.1**	**7.4**	**6.7**	**5.9**	**4.9**	**4.1**

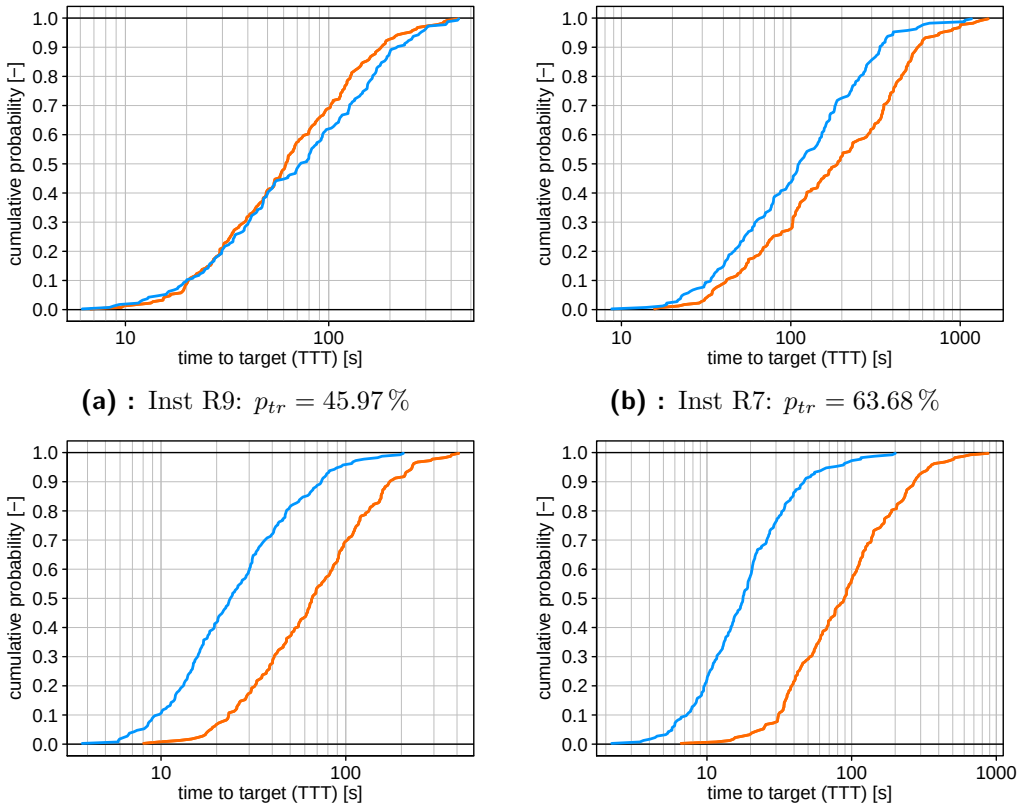
Table 4.3: *Time-limits* results on instances with 1000 customers. All times are in seconds.

to get anywhere nearby it. This behavior could be neglected by generating a purely greedy solution at first, saving it as the incumbent, and then run GILS-RVND as usual. Note that this improvement would merely change the behavior of Ms-GVNS, which works with the purely greedy solution from the start, and provides better-than-UB solutions after no more than one second. Even if we considered the improved *reference* as suggested, after 100 seconds Ms-GVNS would still return better solutions which are on average by 1.8 % closer to c_{best} than solutions returned by *the reference*. Based on these observations, we claim that Ms-GVNS is superior to GILS-RVND in scenarios with up to 1000 customers and a hard limit on computational time ranging from 1 to 100 seconds.

The complementary results — *fixed-iters* and *TTT-plots* — are summarized in Table 4.4 for instances with 10, 20, 50, 100, and 200 customers. For instances with up to 50 customers, both methods found the optimal solution in all executions over all instances and sizes, i.e., the average values of %bG and %mG are both exactly zero. For 100 and 200-customer instances, GILS-RVND's %mG is slightly better than Ms-GVNS's, as well as is %bG for 200-customer because Ms-GVNS did not find the best-known solution for two instances (R1 and R2). On the other hand, the Time improvement of Ms-GVNS over GILS-RVND is 37 %, 82 %, 118 %, 136 %, and 40 % for the considered sizes, respectively. The average *TTT-plots* results are favorable towards Ms-GVNS. The average improvement in reaching the target solution is 15%, 38%, 64%, 91%, and 73%, respectively, and the average probabilities of returning the target solution in time before *the reference* are 62 %, 71 %, 69 %, 71 %, and 78 % respectively.

The full *fixed-iters* and *TTT-plots* results for 500-customer instances are shown in Table 4.5. Here, *the proposed* provides better gaps; however, the run-time of classical *fixed-iters* experiments is worse, on average, by about 43% when compared to *the reference*. It seems like the roles of the two algorithms have switched as we consider large problems. This observation is caused by two factors specific to the *fixed-iters* computational context: (i) the longer the method runs, the better solution usually returns at the end, and (ii) the running time of the method depends on the number of tries to improve the incumbent solution before it gives up and moves to the next iteration. Regarding the latter, GILS-RVND gives up after $\min(size(i), 100)$, and Ms-GVNS after $\lceil size(i)/5 \rceil \cdot |\mathcal{P}|$ tries on instance i. In other words, for instances with 100, 200, and 500 customers, *the reference* has the number of tries fixed to 100, but *the proposed* performs 63, 123, and 303 tries, respectively. To conclude, Ms-GVNS is more thorough for larger instances than GILS-RVND, which results in better gaps and longer running times, but for smaller instances, it is the opposite. Thus, it is hard to make an absolute statement about which method performs better in general.

Fixed-iters computational context may provide useful, informative results, but it is not suitable for the absolute comparison of the methods. Rather, it is more



(a) : Inst R9: $p_{tr} = 45.97\,\%$

(b) : Inst R7: $p_{tr} = 63.68\,\%$

(c) : Inst R13: $p_{tr} = 80.68\,\%$

(d) : Inst R15: $p_{tr} = 91.22\,\%$

Figure 4.2: TTT-plots for Ms-GVNS (blue) and GILS-RVND (orange) on four instances with 500 customers.

relevant to compare the methods according to their p_{pr} in the *TTT-plots* context, as discussed in Sec. 1.1, 1.3.4, and 1.3.5. In terms of probability p_{pr}, *the proposed* is almost always better than *the reference*, except for two 500-customer instances.

To complement Table 4.5, we show full TTT-plots for four instances: one with the lowest p_{pr} (Fig. 4.2a), one with p_{pr} close to median (Fig. 4.2b), and two with the highest p_{pr} (Fig. 4.2c, and 4.2d). In the end, the last table also shows in **bold** four newly found best-known solutions of instances TRP-S500-R2, TRP-S500-R8, TRP-S500-R11, and TRP-S500-R19.

	GILS-RVND				Ms-GVNS				
Size	$\overline{\%\text{bG}}$	$\overline{\%\text{mG}}$	$\overline{\text{Time}}$	$\overline{\text{TTT}}$	$\overline{\%\text{bG}}$	$\overline{\%\text{mG}}$	$\overline{\text{Time}}$	$\overline{\text{TTT}}$	$\overline{\%p_{pr}}$
11	0.000	0.000	$1.3 \cdot 10^{-3}$	$5.2 \cdot 10^{-5}$	0.000	0.000	$9.0 \cdot 10^{-4}$	$4.5 \cdot 10^{-5}$	61.9
21	0.000	0.000	$1.4 \cdot 10^{-2}$	$7.1 \cdot 10^{-4}$	0.000	0.000	$7.4 \cdot 10^{-3}$	$5.2 \cdot 10^{-4}$	70.5
51	0.000	0.000	$2.6 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$	0.000	0.000	$1.2 \cdot 10^{-1}$	$1.5 \cdot 10^{-2}$	69.0
101	0.000	0.000	$3.3 \cdot 10^{0}$	$9.8 \cdot 10^{-2}$	0.000	0.002	$1.4 \cdot 10^{0}$	$5.1 \cdot 10^{-2}$	70.9
201	0.000	0.035	$2.9 \cdot 10^{1}$	$2.0 \cdot 10^{0}$	0.004	0.080	$2.1 \cdot 10^{1}$	$1.1 \cdot 10^{0}$	67.7

Table 4.4: Average *Fixed-iters* and *TTT-plots* results on instances with 10, 20, 50, 100, and 200 customers. All times are in seconds.

			GILS-RVND				Ms-GVNS						
Inst	c_{best}	c_{goal}	%bG	%mG	Time	TTT	Best	%bG	Mean	%mG	Time	TTT	$\%p_{pr}$
R1	1841210	1859799	0.01	0.80	615.3	371.9	1848215	0.38	1852248.7	0.60	952.3	282.4	56.15
R2	1815664	1834733	0.05	0.41	608.8	112.5	**1815478**	−0.01	1818765.5	0.17	806.1	58.0	71.30
R3	1826738	1851374	0.35	0.69	627.0	84.9	1830510	0.21	1833469.3	0.37	939.6	59.9	62.76
R4	1802921	1827358	0.35	0.72	635.7	128.8	1803003	0.00	1811107.7	0.45	934.8	82.4	63.45
R5	1821250	1842214	0.15	0.70	556.8	185.8	1823213	0.11	1829110.9	0.43	837.5	111.7	62.76
R6	1782731	1804486	0.22	0.46	615.4	98.0	1786903	0.23	1790935.4	0.46	806.7	48.0	72.67
R7	1846251	1866478	0.09	0.63	666.6	275.2	1847322	0.06	1853758.1	0.41	936.6	165.2	63.68
R8	1819636	1839054	0.07	0.53	618.9	248.4	**1818621**	−0.06	1826246.4	0.36	893.3	154.9	62.40
R9	1729796	1751157	0.23	0.42	599.0	86.1	1734166	0.25	1739277.9	0.55	794.9	101.1	45.97
R10	1761174	1780368	0.09	0.35	624.1	148.4	1762984	0.10	1766260.3	0.29	874.9	102.8	62.21
R11	1797771	1815859	0.01	0.21	530.6	101.1	**1797111**	−0.04	1801042.4	0.18	825.9	49.0	72.26
R12	1774452	1792196	0.00	0.53	575.4	151.9	1775230	0.04	1780101.6	0.32	948.5	161.8	48.70
R13	1863905	1892435	0.53	0.76	625.2	88.0	1865963	0.11	1870876.1	0.37	817.7	34.2	80.68
R14	1796129	1817162	0.17	0.53	667.7	164.6	1798223	0.12	1802641.2	0.36	888.1	67.4	73.77
R15	1784919	1809056	0.35	0.71	621.4	125.5	1786988	0.12	1791543.4	0.37	780.7	25.2	91.22
R16	1804392	1828289	0.32	0.67	637.6	103.5	1806297	0.11	1809260.2	0.27	879.1	45.6	75.20
R17	1819909	1844005	0.32	0.80	594.0	112.9	1823132	0.18	1825989.3	0.33	926.1	54.4	71.85
R18	1825615	1844525	0.04	0.42	650.1	104.0	1825659	0.00	1829417.4	0.21	904.3	77.1	60.15
R19	1776855	1797040	0.13	0.33	598.1	105.2	**1775030**	−0.10	1779258.1	0.14	911.2	59.9	68.54
R20	1820168	1839021	0.04	0.57	623.4	213.8	1822641	0.14	1828615.0	0.46	942.5	165.4	56.79
avg	-	-	**0.18**	**0.56**	**614.6**	**150.5**	-	**0.10**	-	**0.36**	**880.0**	**95.3**	**66.13**

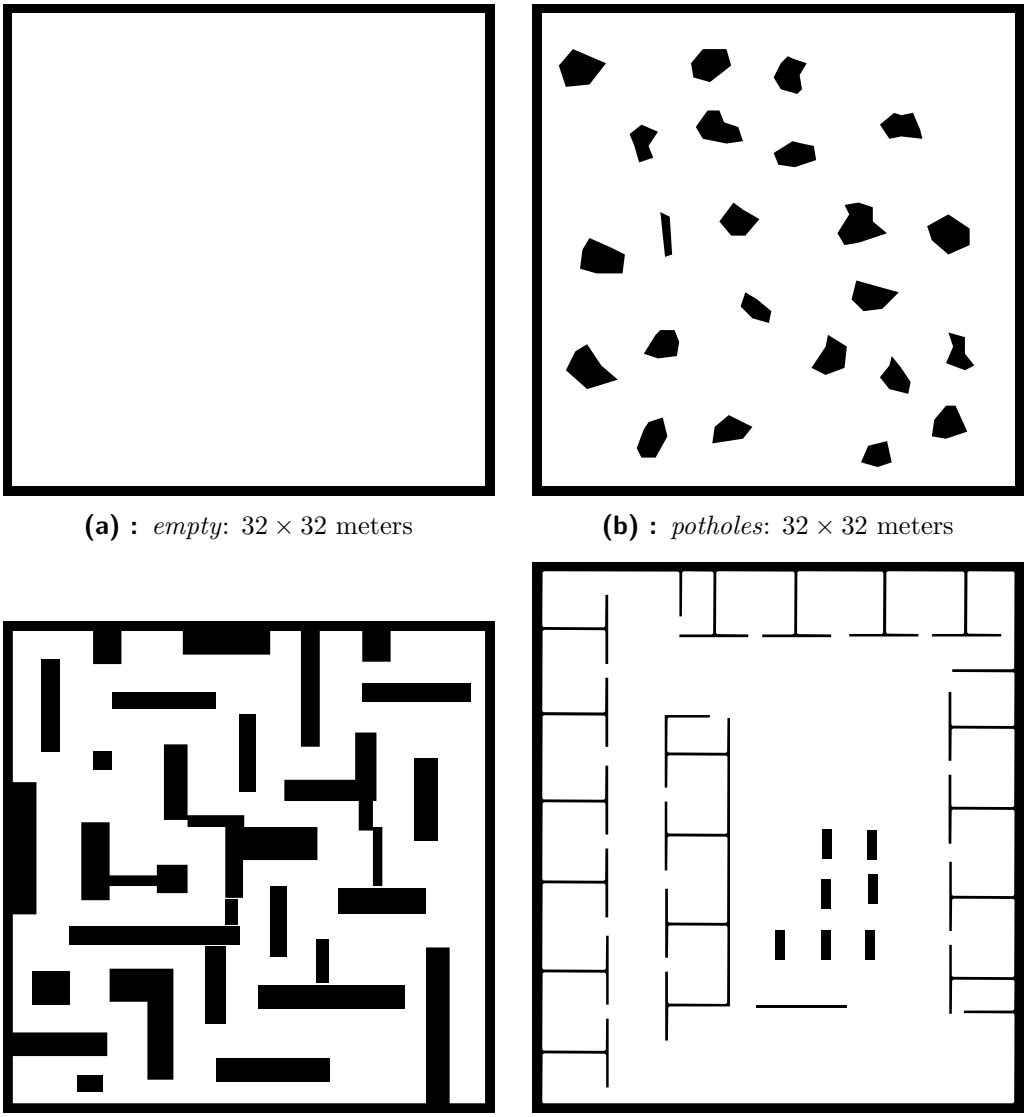
Table 4.5: *Fixed-iters* and *TTT-plots* results on instances with 500 customers. All times are in seconds.

4.2 Mobile search

This section is dedicated to evaluating algorithms described in Sec. 3.2, 3.3, and 3.4 on instances of the *mobile search*. For the *mobile search*, we design and implement in `C++` an entire framework composed of custom ROS packages. ROS — *Robot Operating System* [86] — is a collection of software tools for robot software development. It provides libraries, visualizers, hardware abstraction, device drivers, message-passing and more. We use *Melodic Morenia* distribution of ROS. For the package management we use *Catkin Command Line Tools*[1] — a software tool designed by the ROS community to efficiently build numerous inter-dependent, but separately developed, *CMake* projects. The collection of algorithms that together make the *mobile search* framework is partially implemented from scratch and partially with the use of external libraries. The used libraries include the following: *Boost `C++` Libraries* [87] (for graph algorithms and various helper tools), *Clipper*[2] (for polygon clipping and offsetting), *Triangle* [11] (for *Delaunay triangulations*), *Cairo*[3] (for drawing), *LKH-3* [81] (for solving the TSP), and *PolyPartition*[4] (for convex polygon partitioning). All experiments described in this section are executed on a personal computer with Intel® Core™ i5-7300HQ CPU (2.50 GHz), 8 GB of RAM, and Ubuntu 18.04.1 LTS.

As part of the evaluation, we consider a realistic simulation performed with a commercial robot — *TurtleBot3* [88] — a small ROS-based mobile robot with open source software. *TurtleBot3* simulation physics runs in *Gazebo*[5] simulator, which can be delivered as part of ROS, and the robot's navigation utilizes standard ROS navigation packages[6]. Besides the *TurtleBot3* simulation, we also develop a simple ideal stand-alone simulator, i.e., an application outside ROS, which allows simulating the *mobile search* much faster. The ideal simulator is based on the search scheme, Alg. 3.2, where the actual robot's navigation is replaced by a simple idealized one that considers two types of movements: going straight with linear velocity v_{lin} (m/s), and turning on the spot with angular velocity v_{ang} (rad/s). We do our best to make the two considered simulations alike. First, we tune the *TurtleBot3* navigation demo parameters such that the robot's movements resemble the two basic maneuvers considered in our ideal simulation. Then, we estimate proper values for our ideal simulation velocity constants $v_{lin} = 0.171\,\mathrm{m/s}$, $v_{ang} = 0.312\,\mathrm{rad/s}$ based on the *Gazebo* simulations. The constants' estimation is done by dragging the *TurtleBot3* to complete several routes in the *Gazebo* simulation, then traversing the same routes by our ideal simulation, and finally minimizing the mean square error between the recorded times of both simulations.

[1] Available at `https://catkin-tools.readthedocs.io/en/latest/`
[2] Available at `http://www.angusj.com/delphi/clipper.php`
[3] Available at `https://www.cairographics.org/`.
[4] Available at `https://github.com/ivanfratric/polypartition`.
[5] Available at `http://gazebosim.org/`.
[6] Available at `http://wiki.ros.org/navigation`.



(a) : *empty*: 32×32 meters

(b) : *potholes*: 32×32 meters

(c) : *warehouse*: 32×32 meters

(d) : *jari-huge*: 33×37 meters

Figure 4.3: The environments used in our experiments available at `http://imr.ciirc.cvut.cz/planning/maps.xml`.

In our *mobile search* experiments, we consider four distinct environments: *empty*, *potholes*, *warehouse*, and *jari-huge*, all shown in Fig. 4.3, and three different visibility radii $r_{\mathcal{V}} \in \{\infty, 5, 3\}$ m, where $r_{\mathcal{V}} = \infty$ m represents the unlimited visibility range. All possible pairs of the environments and visibility radii make 12 different *mobile search* instances. However, the combination of the *empty* environment and $r_{\mathcal{V}} = \infty$ m is trivial, i.e., the robot automatically covers the whole environment from the start. Thus, we use only the other 11 non-trivial combinations. The robot's initial position q_0 is at the middle-bottom of each environment, facing up. We assume a circular robot with radius $r_{\mathcal{A}} = 0.4$ meters.

Let us overview all planning algortihms developed in the previous chapter. Regarding the *discretization phase*, we recognize the following variants: DT, KA, DS (Sec. 3.2.1), WR (Sec. 3.2.2), DTF, KAF, DSF, WRF (Sec. 3.2.3), and WRF-DT-F (Sec. 3.2.4). Regarding the *routing phase*, we consider just one general algorithm — our new metaheuristic for the TDP — Ms-GVNS (Sec. 3.3.8). However, this general metaheuristic is, by considering different optimization criterion and improvement calculations, adapted to solve several MLPs modeling the *mobile search* in various ways. The considered MLPs are the following: TDP (Sec. 2.2), ATDP, GSP, AGSP (Sec. 3.4.1), GSP2, and AGSP2 (Sec. 3.4.2). In addition, we recognize four other variants of planning algorithms with replanning: GSP-RP, AGSP-RP, GSP2-RP, and AGSP2-RP (Sec. 3.4.3). These use our ideal simulator to execute the whole search scheme, Alg. 3.2, where the metaheuristic progressively refines the plan as the search progresses, and the resulting overall plan $\boldsymbol{l}^{true}$ is returned at the end. In all individual runs, Ms-GVNS is constrained by a hard limit on computational time $t_{max} = 2$ seconds.

We test selected planning algorithms in the following two steps.

1. **Planning.** Starting with the robot placed in the environment at the initial configuration, we first create the search plan $\boldsymbol{l}^{seq}$. For variants without replanning, it is enough to execute lines 1-3 of Alg. 3.2. For variants with replanning, the whole scheme needs to be executed using our ideal simulator and then $\boldsymbol{l}^{seq}$ is assigned with the resulting $\boldsymbol{l}^{true}$. Here, for planning speed up, we use low sensing frequency $f = 2 \cdot v_{lin} = 0.342\,\text{Hz}$.

2. **Evaluation.** Having the plan prepared, we execute the search simulation according to lines 4-20 of Alg. 3.2 with high sensing frequency $f = 5\,\text{Hz}$ (always ignoring the `RePlan` procedure). As a result, we receive a precise estimation of the expected time t_{exp} to find the searched object, which characterizes the quality of our solution $\boldsymbol{l}^{seq}$. This step can be done in two ways:

   a. we either navigate the robot in our idealized simulator, or
   b. use the *TurtleBot3*'s ROS simulation in *Gazebo* described at the beginning of this section.

   We denote these two options as *ideal* and *ROS* evaluation, respectively.

Let us first deem the *ideal* evaluation. Preliminary tests have shown that *discretization* algorithms with filtering provide better results than their corresponding non-filtering versions. We demonstrate this effect with algorithms DS and DSF. Regarding the rest, we consider only the versions with filtering, i.e., DTF, KAF, WRF, and the hybrid WRF-DT-F. Other preliminary tests have shown that DTF, WRF, and WRF-DT-F, on average, provide better results than DS, DSF, and KAF. Thus, we evaluate the most promising *discretizations* DTF, WRF, WRF-DT-F thoroughly and consider with them all the *routing* versions, i.e., TDP, ATDP, GSP, AGSP, GSP2, AGSP2, GSP-RP, AGSP-RP, GSP2-RP, and AGSP2-RP.

$r_{\mathcal{V}}$	Environment	Abbr.	*Ideal*		*ROS*	
			$t^{\star}_{exp}$ [s]	Met.	$t^{\star}_{exp}$ [s]	Met.
∞	*potholes*	∞-pot	36.78	1	-	-
∞	*warehouse*	∞-war	277.92	1	-	-
∞	*jari-huge*	∞-jar	194.95	2	-	-
5	*empty*	5-emp	333.17	1	327.85	3
5	*potholes*	5-pot	372.33	1	369.26	4
5	*warehouse*	5-war	517.85	3	519.00	1
5	*jari-huge*	5-jar	546.97	1	543.85	1
3	*empty*	3-emp	600.10	2	-	-
3	*potholes*	3-pot	626.05	1	-	-
3	*warehouse*	3-war	718.26	1	-	-
3	*jari-huge*	3-jar	962.67	2	-	-

Table 4.6: The best recorded times $t^{\star}_{exp}$ for all *mobile search* instances and the two types of evaluation. Methods' legend: 1: WRF-DT-F+AGSP-RP, 2: DS+AGSP-RP, 3: WRF-DT-F+AGSP2-RP, 4: DSF+AGSP-RP.

For the less promising *discretizations*, i.e., DS, DSF, and KAF, we consider just the most promising *routing* versions AGSP and AGSP-RP, that we determined, again, based on preliminary results. Totally, we deem $3 \cdot 10 + 3 \cdot 2 = 36$ solution methods, i.e., combinations of *discretization* and *routing* algorithms. For every pair consisting of a *mobile search* instance and a solution method, we perform 20 iterations with different random seeds. Then, we report the percentage mean gap %mG computed as $100 \cdot (\bar{t}_{exp} - t^{\star}_{exp})/t^{\star}_{exp}$, where $\bar{t}_{exp}$ is the mean t_{exp} over the 20 iterations, and $t^{\star}_{exp}$ is the best (lowest) t_{exp} for the considered instance and type of evaluation recorded over all of our experiments. An overview of the best times $t^{\star}_{exp}$ for all considered instances is shown in Tab. 4.6. In the table, an additional column Abbr. shows abbreviations of instances used throughout this section. Columns Met. show which solution methods have found the best times, whereas the legend is included within the table's caption.

The mean gaps, %mGs, for all 36 tested solution methods are shown in Tab. 4.7. The first three columns show the line number and the solution methods' *discretization* and *routing* variants. The next 11 columns are the methods' %mGs for individual instances, and the last column is the %mG averaged over all instances. The table's rows are ordered according to the average performance. Values, which are blue and underlined, are the best results of the respective column, and just blue values are the second best. Overall the best performing method is the one combining WRF-DT-F and AGSP-RP. It provides, on average, the best plans for almost all *mobile search* instances with limited visibility, except one (3-pot), where it provides the second best. For instances with unlimited visibility, it provides intermediate results, whereas the supremacy is distributed over a wider spectrum of other methods.

	Discretization	Routing	Instances											avg
			∞-pot	∞-war	∞-jar	5-emp	5-pot	5-war	5-jar	3-emp	3-pot	3-war	3-jar	
1	WRF-DT-F	AGSP-RP	9.6	8.7	24.8	2.7	5.1	4.7	4.7	4.7	3.5	2.5	4.8	**6.9**
2	DTF	AGSP-RP	2.5	0.8	25.2	2.8	8.0	14.9	7.5	5.6	2.2	3.1	5.9	**7.1**
3	WRF-DT-F	GSP-RP	9.4	10.5	31.3	3.9	7.3	9.1	8.3	8.7	7.7	5.6	7.7	**10.0**
4	WRF-DT-F	AGSP2-RP	8.2	13.1	33.3	8.1	12.2	6.4	7.3	7.4	5.7	7.0	6.1	**10.4**
5	DTF	AGSP2-RP	2.8	1.6	32.7	11.4	9.3	15.3	7.8	9.2	7.1	12.6	6.1	**10.6**
6	DTF	GSP-RP	2.5	0.7	50.3	4.6	7.3	17.0	9.6	9.4	7.5	7.5	10.2	**11.5**
7	WRF	AGSP-RP	26.1	12.7	25.9	11.6	10.7	6.3	6.9	9.8	7.0	5.1	8.8	**11.9**
8	DSF	AGSP-RP	27.5	21.4	28.1	11.4	7.5	8.1	8.9	6.2	6.8	4.8	6.9	**12.5**
9	DS	AGSP-RP	47.9	21.6	24.3	9.2	6.8	7.1	9.2	6.0	6.5	4.7	5.3	**13.5**
10	KAF	AGSP-RP	31.2	14.2	11.9	14.5	8.1	12.6	15.0	12.8	5.2	6.9	22.4	**14.1**
11	WRF-DT-F	GSP2-RP	9.6	14.0	50.3	11.1	17.1	9.4	10.3	12.6	11.8	8.2	8.9	**14.8**
12	WRF	GSP-RP	27.6	13.1	32.7	13.9	14.0	7.0	11.3	13.0	11.9	7.4	12.3	**14.9**
13	DTF	GSP2-RP	2.8	10.3	54.6	14.9	11.2	15.3	9.6	13.9	13.8	12.8	10.2	**15.4**
14	WRF-DT-F	AGSP	14.2	24.3	36.6	11.0	17.8	10.4	19.2	14.0	17.9	8.9	11.8	**16.9**
15	WRF	AGSP2-RP	47.4	22.2	32.2	15.2	14.6	10.7	9.0	12.1	9.7	7.3	9.8	**17.3**
16	WRF-DT-F	GSP	14.7	26.4	36.5	13.9	21.5	13.6	20.0	19.8	22.5	11.2	14.9	**19.5**
17	WRF-DT-F	ATDP	12.9	39.0	32.4	11.6	27.8	15.8	20.0	13.8	18.7	11.3	13.4	**19.7**
18	WRF	AGSP	31.6	30.7	38.0	22.4	21.0	12.6	19.3	18.5	15.5	8.3	17.8	**21.4**
19	WRF	GSP2-RP	51.2	28.0	45.7	20.3	19.3	10.7	12.7	15.9	14.2	9.2	13.0	**21.8**
20	DTF	AGSP	7.7	16.2	88.3	12.4	21.5	17.4	21.0	15.7	20.4	10.6	14.9	**22.4**
21	WRF-DT-F	TDP	14.5	38.9	35.6	14.6	31.6	16.4	25.0	18.9	22.2	13.1	17.8	**22.6**
22	WRF-DT-F	AGSP2	15.9	38.5	49.9	14.5	28.9	17.1	22.4	22.0	23.3	11.7	15.6	**23.6**
23	WRF	GSP	32.0	33.4	38.1	23.4	25.6	13.0	22.5	24.0	22.1	10.8	20.5	**24.1**
24	DSF	AGSP	31.3	46.8	58.6	21.1	18.1	15.4	20.6	16.2	14.9	9.8	15.8	**24.4**
25	WRF	ATDP	46.8	44.7	32.7	22.4	26.0	13.4	22.3	19.3	17.7	9.9	20.6	**25.1**
26	DTF	GSP	7.7	25.4	93.8	17.5	25.0	17.2	22.8	19.8	27.3	14.5	18.3	**26.3**
27	DTF	ATDP	7.7	24.4	111.1	12.4	21.5	20.7	24.4	17.6	23.5	11.1	15.7	**26.4**
28	WRF-DT-F	GSP2	19.0	39.6	50.3	18.0	32.5	16.9	24.9	26.0	25.5	16.2	21.7	**26.4**
29	KAF	AGSP	48.5	42.7	38.0	27.6	23.7	22.3	30.4	22.4	13.1	16.0	31.9	**28.8**
30	WRF	AGSP2	51.6	46.6	50.7	25.9	27.2	18.4	22.6	23.9	21.3	12.9	20.4	**29.2**
31	WRF	TDP	43.5	50.9	34.6	25.6	33.9	16.5	29.1	25.7	24.3	11.9	27.4	**29.4**
32	DTF	AGSP2	10.2	25.9	116.4	23.3	21.5	19.9	26.5	25.6	24.7	16.4	19.3	**30.0**
33	DTF	TDP	7.7	24.4	116.4	19.4	37.9	19.3	34.7	19.7	31.6	15.2	18.4	**31.3**
34	WRF	GSP2	56.3	53.5	50.8	27.4	27.8	20.9	28.2	27.4	25.2	15.7	26.6	**32.7**
35	DTF	GSP2	15.0	25.9	116.4	25.0	35.0	19.9	25.3	31.5	29.7	19.0	22.4	**33.2**
36	DS	AGSP	65.5	63.8	113.4	22.7	21.3	19.0	26.8	17.5	19.6	14.8	18.5	**36.6**

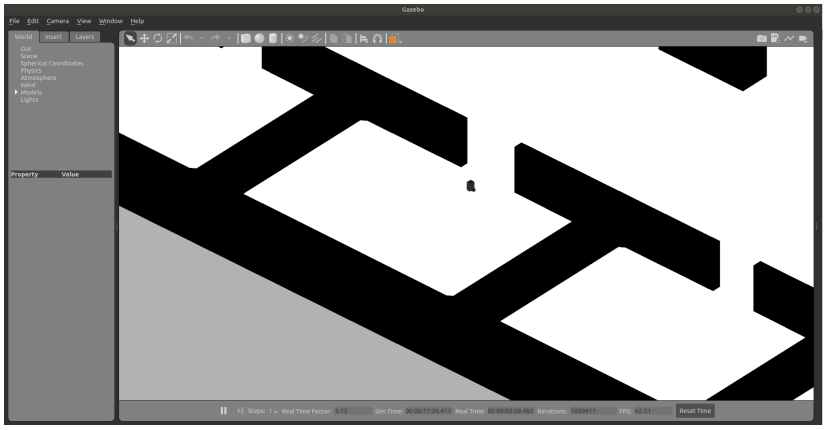
Table 4.7: *Ideal* evaluation results. Table shows %mG for all tested solution methods and considered *mobile search* instances.

Let us now discuss individual influences of different *discretization* and *routing* algorithms on the quality of resulting plans. For easier orientation, we present Tab. 4.8 that extracts just the relevant rows from the full Tab. 4.7. The first part of the table (above the middle line) shows the influence of *routing* variants, whereas the *discretization* is fixed on WRF-DT-F. Firstly note that considering turning angles always improves the plans as it can be seen on pairs TDP → ATDP, GSP → AGSP, etc. Next, one-dimensional weights also positively influence the results as seen on pairs TDP → GSP, ATDP → AGSP, etc. Surprisingly, two-dimensional weights have the opposite effect and significantly worsen the plans according to pairs GSP → GSP2, AGSP → AGSP2, etc. The negative effect is most likely caused by the way we compute the weights. We compute the two-dimensional weights straightforwardly as areas of $\mathcal{V}(l_i, l_j)$, which is analogous to how we compute the one-dimensional weights, as described in Sec. 3.4.2. However, this is probably not the best way, as we weight the beginning of the shortest route from l_i to l_j the same way as the end, which does not match the general aims of *mobile search*. In *mobile search*, portions of the environment seen early during the search are more significant than those seen later, as discussed before on several occasions. This approach might be as well applied to the subproblem of weighting the shortest paths. We reserve this idea for future research. For now, let us continue analyzing the current results. Replanning always improves the plans according to pairs GSP → GSP-RP, AGSP → AGSP-RP, etc., and notably, all solution methods with replanning are superior to those without it. To conclude, almost all suggested extensions of the TDP actually improve the *mobile search* results, except one — the two-dimensional weights. However, we plan to advance the weights' computations in later research to investigate its potential more properly. Overall, the currently best way to discretely model the *mobile search*, based on our results, is by utilizing AGSP-RP.

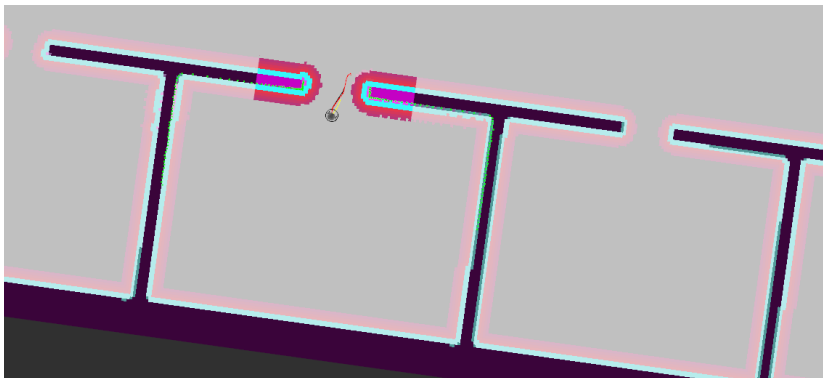
The quality of a solution to the *mobile search* is also dependent on how the environment is discretized. The effect of different *discretizations* can be easily observed in the second part of Tab. 4.8 (bellow the middle line), where the *routing* is fixed to AGSP-RP. The best properties with respect to the *mobile search* has the hybrid WRF-DT-F *discretization* efficiently combining sophisticated WRP approach, straightforward but dense triangulation, and the proposed filtering procedure. The standalone related methods, DTF and WRF, are the second and third best, respectively. However, the gap between DTF and WRP is surprisingly large. An interpretation of this observation might be the following. The WRF strengths mostly show off around dense obstacle areas such as peeking into rooms in the *jari-huge* environment. Despite the resulting plans being visually appealing, the dense obstacle areas are usually visited at later stages of the search and therefore have just a minor effect on the solution's quality. Conversely, in large spaces where there are overlaps between MCCSs, the *LKH-3* may not find the best ordering w.r.t. to the *mobile search*, which in turn leads to poor optimization of the l_i locations in the MCCSs. Plus, of course, the *LKH-3* solves TSP and not TDP. As a result, the combination of dense coverage and efficient filtering is superior to the WRP sophisticated approach, especially in open areas that

	Discretization	Routing	Instances											avg
			∞-pot	∞-war	∞-jar	5-emp	5-pot	5-war	5-jar	3-emp	3-pot	3-war	3-jar	
1	WRF-DT-F	AGSP-RP	9.6	8.7	24.8	2.7	5.1	4.7	4.7	4.7	3.5	2.5	4.8	**6.9**
3		GSP-RP	9.4	10.5	31.3	3.9	7.3	9.1	8.3	8.7	7.7	5.6	7.7	**10.0**
4		AGSP2-RP	8.2	13.1	33.3	8.1	12.2	6.4	7.3	7.4	5.7	7.0	6.1	**10.4**
11		GSP2-RP	9.6	14.0	50.3	11.1	17.1	9.4	10.3	12.6	11.8	8.2	8.9	**14.8**
14		AGSP	14.2	24.3	36.6	11.0	17.8	10.4	19.2	14.0	17.9	8.9	11.8	**16.9**
16		GSP	14.7	26.4	36.5	13.9	21.5	13.6	20.0	19.8	22.5	11.2	14.9	**19.5**
17		ATDP	12.9	39.0	32.4	11.6	27.8	15.8	20.0	13.8	18.7	11.3	13.4	**19.7**
21		TDP	14.5	38.9	35.6	14.6	31.6	16.4	25.0	18.9	22.2	13.1	17.8	**22.6**
22		AGSP2	15.9	38.5	49.9	14.5	28.9	17.1	22.4	22.0	23.3	11.7	15.6	**23.6**
28		GSP2	19.0	39.6	50.3	18.0	32.5	16.9	24.9	26.0	25.5	16.2	21.7	**26.4**
1	WRF-DT-F	AGSP-RP	9.6	8.7	24.8	2.7	5.1	4.7	4.7	4.7	3.5	2.5	4.8	**6.9**
2	DTF		2.5	0.8	25.2	2.8	8.0	14.9	7.5	5.6	2.2	3.1	5.9	**7.1**
7	WRF		26.1	12.7	25.9	11.6	10.7	6.3	6.9	9.8	7.0	5.1	8.8	**11.9**
8	DSF		27.5	21.4	28.1	11.4	7.5	8.1	8.9	6.2	6.8	4.8	6.9	**12.5**
9	DS		47.9	21.6	24.3	9.2	6.8	7.1	9.2	6.0	all 6.5	4.7	5.3	**13.5**
10	KAF		31.2	14.2	11.9	14.5	8.1	12.6	15.0	12.8	5.2	6.9	22.4	**14.1**

Table 4.8: *Ideal* evaluation results — overview of relevant results, i.e., selected rows extracted from previous Tab. 4.7, to study the individual influence of different *discretization* and *routing* algorithms.



(a) : *TurtleBot3* in *Gazebo* simulator.



(b) : Robot data in *RViz*, visualiser for ROS.

Figure 4.4: Example screenshots from the ROS simulation with *TurtleBot3* inside the *jari-huge* environment.

appear in environments *empty* and *potholes*. The proposed method, WRF, would be better justified in solving the WRP, which is something we reserve for future research. Finally, the positive effect of filtering can be seen on DS and DSF. The improvement shown in Tab. 4.8 for AGSP-RP appears minor, but the improvement in the case of AGSP (without replanning), shown in the previous Tab. 4.7, is significant.

The *ideal mobile search* results discussed so far are only relevant to a robot with simplified kinematics, as assumed by our ideal simulator. To be more realistic, we also include the *ROS* evaluation with *TurtleBot3* in *Gazebo* simulator, whose example screenshots are shown in Fig. 4.4. The purpose of these second simulations is to verify that our ideal simulations are, to some extent, relevant also to more realistic robots. However, since *Gazebo* simulation is much more time demanding, we have to limit the scale of the performed experiments. We consider only the four instances with $r_{\mathcal{V}} = 5$ (i.e., 5-emp, 5-pot, 5-war, 5-jar), a subset of planning algorithms, and just five iterations for each instance-algorithm pair. The results obtained by the realistic *ROS* simulations are compared to the *ideal* simulation

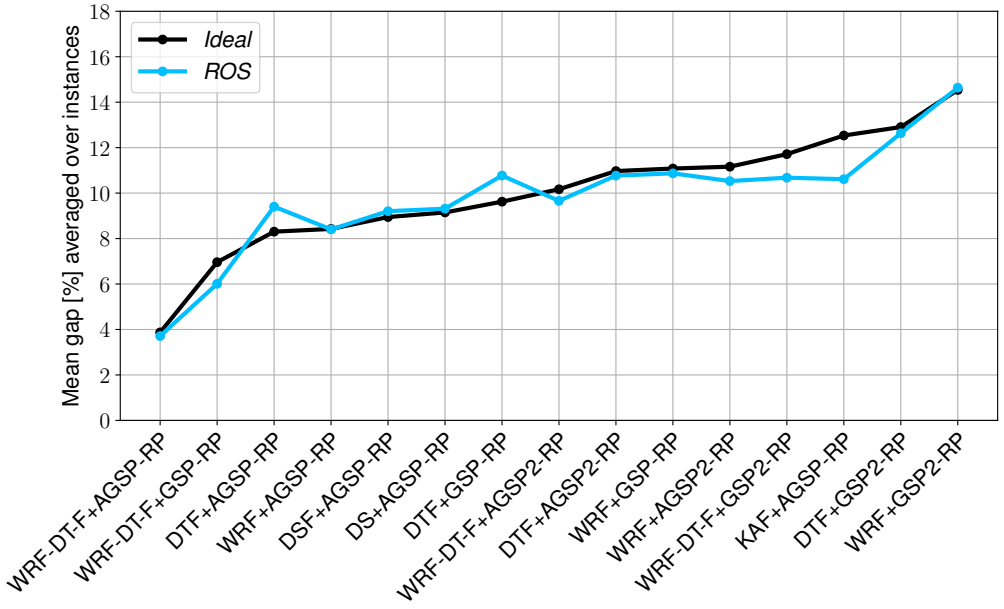
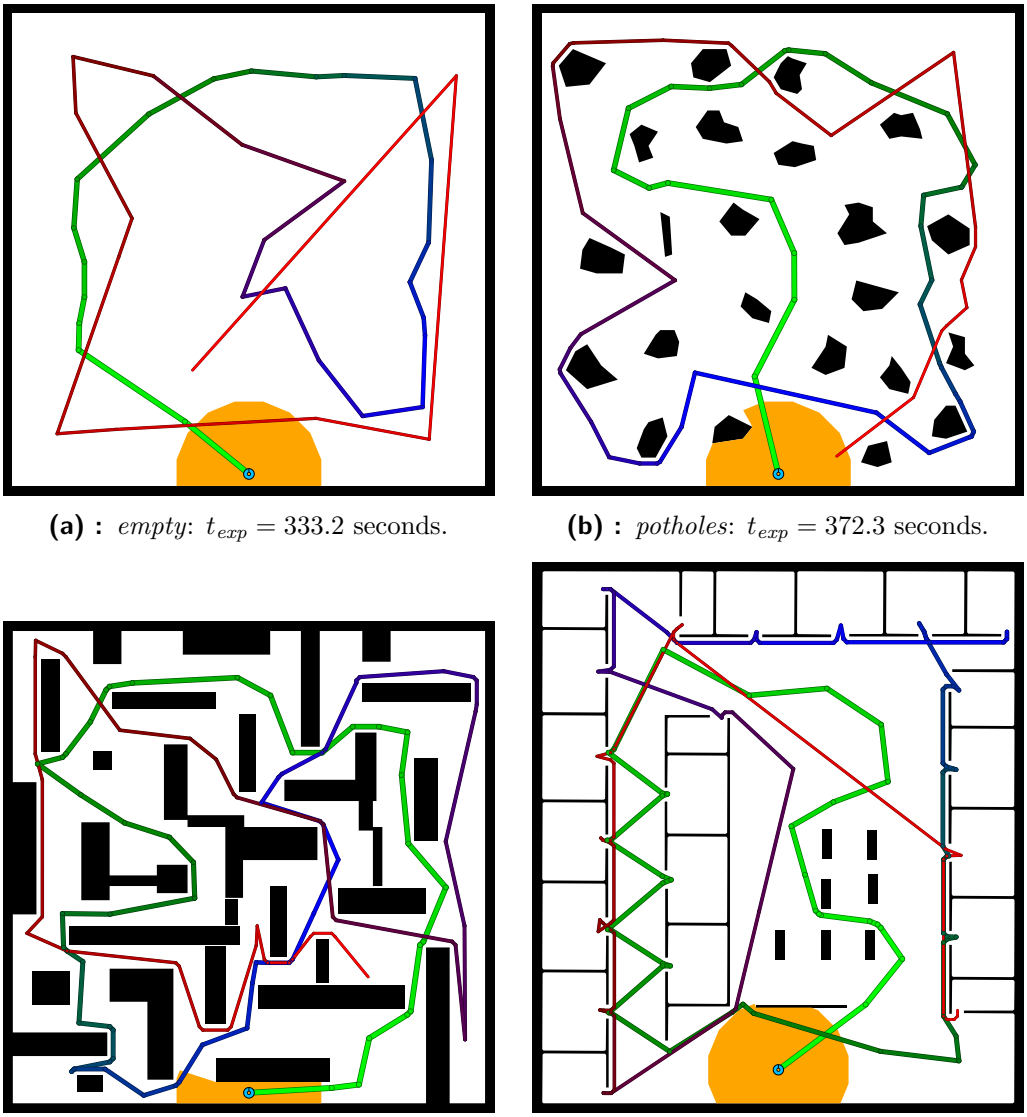


Figure 4.5: Comparison of the *ideal* and *ROS* evaluation.

results of identical scale in Fig. 4.5. The figure displays the %mGs based on five iterations averaged over the four considered instances. We can see that the results of both types are nearly identical in the case of the best solution method combining WRF-DT-F and AGSP-RP and about half of the other methods. For the second half, some deviations can be observed. These are most likely caused by imprecise modeling of the robot's kinematics in our ideal simulator and also by occasional errors during the *ROS* simulation. Especially the *dynamic window approach* algorithm [89], which is used as the default local planer for *TurtleBot3*, has sometimes troubles with simple maneuvers such as turning the robot on the spot by 180 degrees. Despite these inconveniences, the overall trends of both evaluation types in Fig. 4.5 correlate. To prove our results even more relevant to real robots, we might introduce some improvements concerning the simulations in the future. For example, our simulator, as well as our *mobile search* formulations, could consider more of the robot's kinematics, such as acceleration. Regarding the *ROS* simulations, we might better tune the navigation parameters or use an entirely different local planner for *TurtleBot3*.

Let us conclude the analysis of algorithms' performance with respect to the quality of the produced plans. The best plans are, on average, constructed by a solution method composed of WRF-DT-F *discretization* algorithm and replanning scheme where the Ms-GVNS general metaheuristic iteratively solves and refines the discretized version of the *mobile search* modeled as AGSP. Note, that this observation is consistent with both evaluation types (*ideal* and *ROS*). The best plans for all environments and $r_{\mathcal{V}} = 5$ produced by the leading method are shown in Fig. 4.6. The robot stands at its initial configuration in the figure, and the currently seen region of the environment is shown in orange. The plan is depicted



(a) : *empty*: $t_{exp} = 333.2$ seconds.

(b) : *potholes*: $t_{exp} = 372.3$ seconds.

(c) : *warehouse*: $t_{exp} = 521.2$ seconds.

(d) : *jari-huge*: $t_{exp} = 547.0$ seconds.

Figure 4.6: The best plans for all environments and $r_{\mathcal{V}} = 5$ produced by the WRF-DT-F+AGSP-RP solution method.

as a route in the environment that changes its color and thickness for easier visual tracking. It starts as green and thick, turns to blue in the middle, and ends as red and thin. All the plans have a common feature. It looks as they guide the robot to always travel two or more rounds around the environments. The first round is to see large areas very early but is quite rough, meaning it leaves behind small uncovered portions distributed evenly over the environment. The first round is crucial for good result w.r.t. the expected time t_{exp} to find the searched object. The other rounds cover the left-overs, which usually have a small probabilistic chance to exhibit the searched object and therefore have just an insignificant impact on the resulting t_{exp}. At first sight, we can see that those plans would

Symbol	Meaning
cpu_{dis}	Discretization CPU time.
n	The number of generated locations.
cpu_{pre}	Structure pre-computing CPU time.
cpu_{pla}	The first *routing* CPU time.
cpu_{dpp}	$= cpu_{dis} + cpu_{pre} + cpu_{pla}$
cpu_{rep}	Total replanning CPU time.
cpu_{dppr}	$= cpu_{dis} + cpu_{pre} + cpu_{pla} + cpu_{rep}$
t_{end}	Total plan execution time (in simulation time).
t_{exp}	The expected time to find the object (in simulation time).
%mG	$= 100 \cdot (t_{exp} - t^{\star}_{exp})/t^{\star}_{exp}$ (for $t^{\star}_{exp}$ see Tab. 4.6)

Table 4.9: Extended results: the legend.

not be nearby optimum for the WRP, where we only wish to cover the whole environment at the earliest. This observation is in line with the discussion in Sec. 2.1.4.

The remainder of this section shows extended *ideal* results of the best algorithm WRF-DT-F+AGSP-RP and some related variants. The extended results include computational (CPU) times of several algorithms' parts, numbers of generated locations by the *discretization*, plan execution times &c. The complete list of record labels is shown in Fig. 4.9. We explain how the new records particularly relate to the search scheme, Alg. 3.2, next. CPU time cpu_{dis} is the run-time of the `Discretize` procedure, and $n = |\boldsymbol{l}^{set}|$ is the cardinality of its output. CPU time cpu_{pre} is when our program prepares the instance of the *routing* problem, which is then solved by the Ms-GVNS metaheuristic. The instance preparation comprises computing the shortest paths between all pairs of cover-locations and determining the costs and weights that define the discretized optimization problem. In other words, cpu_{pre} is the time between procedures `Discretize` and `Plan`. CPU time cpu_{pla} is the time of the optimization, i.e., when Ms-GVNS solves the *routing* problem, or the run-time of procedure `Plan`. The optimization time is, in all cases, constrained by a hard time limit $t_{max} = 2$ seconds, as noted before. CPU time cpu_{dpp} is the total time of planning for algorithms without replanning. CPU time cpu_{rep} is the computational time between lines 4-20 of Alg. 3.2 for variants with replanning. CPU time cpu_{dppr} is the total time of planning for algorithms with replanning. Simulation time t_{end} is the last tick recorded at line 16 (during the evaluation step), i.e., the time that WRP would optimize.

The extended results for methods WRF-DT-F+{AGSP-RP, GSP-RP, AGSP2-RP, GSP2-RP} are shown in Tab. 4.10. In the table, all values are averaged over 20 iterations of the *ideal* simulation, and the last column shows the average over all considered instances. Records cpu_{dis}, n, and cpu_{pre} are the same for the methods so we show them only once at the first three rows of the table. Note

that the best overall method shown at the top produces plans which take the longest to execute; however, the average expected time t_{exp} is the lowest. Once again, this observation demonstrates the difference between the *mobile search* and the WRP. Also note, that the average CPU time of planning without replanning, cpu_{dpp}, is about five seconds for all methods. On the other hand, the average total time of planning, replanning included, cpu_{dppr}, is about two-times higher for the best method compared to the one that considers GSP instead of the more complex AGSP. However, this may not be a problem since, in a real application, the replanning can be performed online as the search progresses, so the robot would only need to wait five seconds for the first plan.

The extended results for methods {DTF, WRF, DSF}+AGSP-RP are shown in Tab. 4.11. Here, we remark just one final observation. The method that produces the second-best average plans, i.e., the one with DTF *discretization*, can provide the first plan in only 1.7 seconds, on average. That is 3-times faster than the one with sophisticated WRF-DT-F *discretization* producing the best average plans.

Discretization	Routing		Instances											
			∞-pot	∞-war	∞-jar	5-emp	5-pot	5-war	5-jar	3-emp	3-pot	3-war	3-jar	**avg**
WRF-DT-F		cpu_{dis}	8.5	5.1	7.2	0.4	1.7	1.3	2.2	2.0	3.5	2.5	5.0	**3.6**
		n	15.8	27.9	23.2	35.5	40.3	43.8	53.4	83.8	87.7	76.8	122.8	**55.6**
		cpu_{pre}	0.5	0.1	0.7	0.3	0.5	0.2	0.5	1.4	1.1	0.4	1.4	**0.6**
WRF-DT-F	AGSP-RP	cpu_{pla}	0.0	0.0	0.0	0.1	0.2	0.2	0.7	2.0	2.0	2.0	2.0	**0.8**
		cpu_{dpp}	9.0	5.3	8.0	0.8	2.4	1.7	3.4	5.3	6.6	4.9	8.4	**5.1**
		cpu_{rep}	0.0	0.2	0.2	0.7	1.5	2.1	5.8	52.7	61.3	33.3	120.8	**25.3**
		cpu_{dppr}	9.0	5.5	8.2	1.5	3.9	3.9	9.2	58.0	67.9	38.2	129.2	**30.4**
		t_{end}	846.3	1424.7	1639.2	1277.1	1475.3	1954.6	2558.3	2173.6	2238.8	2493.8	4207.5	**2026.3**
		t_{exp}	40.3	302.1	243.2	342.1	391.5	542.0	572.7	628.2	647.7	736.0	1008.6	**495.9**
		%mG	9.6	8.7	24.8	2.7	5.1	4.7	4.7	4.7	3.5	2.5	4.8	**6.9**
WRF-DT-F	GSP-RP	cpu_{pla}	0.0	0.0	0.0	0.0	0.1	0.1	0.2	1.1	1.3	0.8	2.0	**0.5**
		cpu_{dpp}	9.0	5.3	8.0	0.7	2.3	1.6	3.0	4.5	5.9	3.7	8.4	**4.7**
		cpu_{rep}	0.0	0.1	0.1	0.3	0.7	0.9	2.3	19.1	24.2	10.7	76.0	**12.2**
		cpu_{dppr}	9.0	5.4	8.1	1.1	3.0	2.5	5.3	23.6	30.0	14.4	84.4	**17.0**
		t_{end}	844.6	1482.9	1681.2	1267.5	1415.9	1916.6	2376.6	2135.6	2273.3	2493.3	4234.0	**2011.0**
		t_{exp}	40.2	307.0	255.9	346.2	399.5	565.2	592.4	652.3	674.3	758.8	1036.4	**511.7**
		%mG	9.4	10.5	31.3	3.9	7.3	9.1	8.3	8.7	7.7	5.6	7.7	**10.0**
WRF-DT-F	AGSP2-RP	cpu_{pla}	0.0	0.0	0.0	0.1	0.2	0.3	0.5	2.0	2.0	1.8	2.0	**0.8**
		cpu_{dpp}	9.0	5.3	8.0	0.8	2.4	1.8	3.4	5.4	6.8	4.8	8.8	**5.1**
		cpu_{rep}	0.3	0.8	1.6	1.2	4.8	4.2	12.4	60.9	86.4	39.2	199.3	**37.4**
		cpu_{dppr}	9.3	6.1	9.7	1.9	7.2	6.0	15.8	66.4	93.1	44.0	208.2	**42.5**
		t_{end}	773.9	1402.9	1317.1	1031.4	1332.8	1714.9	2200.9	1931.4	2085.8	2134.3	3788.3	**1792.2**
		t_{exp}	39.8	314.4	259.8	360.3	417.9	550.7	586.8	644.2	661.9	768.5	1021.5	**511.4**
		%mG	8.2	13.1	33.3	8.1	12.2	6.4	7.3	7.4	5.7	7.0	6.1	**10.4**
WRF-DT-F	GSP2-RP	cpu_{pla}	0.0	0.0	0.0	0.0	0.1	0.1	0.2	1.0	1.3	0.6	2.0	**0.5**
		cpu_{dpp}	9.0	5.3	8.0	0.7	2.3	1.6	3.0	4.4	6.0	3.6	8.7	**4.8**
		cpu_{rep}	0.3	0.7	1.3	0.8	3.8	2.9	9.2	29.3	49.4	20.6	157.5	**25.1**
		cpu_{dppr}	9.3	6.0	9.4	1.5	6.1	4.5	12.3	33.7	55.4	24.2	166.2	**29.9**
		t_{end}	781.7	1387.6	1106.7	999.3	1299.4	1756.2	2082.6	1932.4	2040.6	2096.7	3908.3	**1762.9**
		t_{exp}	40.3	316.8	293.1	370.0	436.0	566.4	603.1	675.7	700.1	776.8	1048.5	**529.7**
		%mG	9.6	14.0	50.3	11.1	17.1	9.4	10.3	12.6	11.8	8.2	8.9	**14.8**

Table 4.10: Extended results for WRF-DT-F+{AGSP-RP, GSP-RP, AGSP2-RP, GSP2-RP}. All times are in seconds.

Discretization	Routing		Instances											
			∞-pot	∞-war	∞-jar	5-emp	5-pot	5-war	5-jar	3-emp	3-pot	3-war	3-jar	**avg**
DTF	AGSP-RP	cpu_{dis}	0.4	0.1	0.4	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.1	**0.1**
		n	16.0	32.0	27.0	38.0	41.0	43.0	55.0	86.0	90.0	84.0	122.0	**57.6**
		cpu_{pre}	0.5	0.1	1.1	0.3	0.5	0.2	0.6	1.4	1.1	0.4	1.4	**0.7**
		cpu_{pla}	0.0	0.1	0.0	0.1	0.2	0.2	0.7	2.0	2.0	2.0	2.0	**0.9**
		cpu_{dpp}	0.9	0.3	1.5	0.5	0.8	0.5	1.4	3.5	3.2	2.5	3.6	**1.7**
		cpu_{rep}	0.0	0.5	0.4	1.1	1.6	2.0	6.4	57.0	69.4	47.2	123.4	**28.1**
		cpu_{dppr}	1.0	0.8	1.9	1.6	2.4	2.5	7.8	60.6	72.5	49.7	127.0	**29.8**
		t_{end}	875.9	1618.6	2367.9	1248.9	1484.2	1952.3	2404.3	2330.6	2400.8	3136.3	4750.7	**2233.7**
		t_{exp}	37.7	280.1	244.0	342.5	402.1	595.1	588.0	633.6	640.1	740.9	1019.8	**502.2**
		%mG	2.5	0.8	25.2	2.8	8.0	14.9	7.5	5.6	2.2	3.1	5.9	**7.1**
WRF	AGSP-RP	cpu_{dis}	8.2	5.1	7.1	0.4	1.6	1.3	2.1	1.9	3.4	2.5	4.9	**3.5**
		n	14.7	26.6	23.4	31.1	38.5	44.0	52.3	79.1	81.0	77.2	118.9	**53.3**
		cpu_{pre}	0.4	0.1	0.7	0.2	0.4	0.2	0.5	1.2	0.9	0.4	1.3	**0.6**
		cpu_{pla}	0.0	0.0	0.0	0.1	0.2	0.3	0.6	2.0	2.0	1.9	2.0	**0.8**
		cpu_{dpp}	8.6	5.2	7.8	0.6	2.3	1.7	3.3	5.1	6.3	4.7	8.2	**4.9**
		cpu_{rep}	0.0	0.2	0.2	0.4	1.4	2.2	5.3	43.9	49.3	30.9	112.3	**22.4**
		cpu_{dppr}	8.6	5.4	8.0	1.0	3.6	3.9	8.6	49.1	55.6	35.6	120.5	**27.3**
		t_{end}	767.3	1468.8	1656.8	1260.5	1501.0	1911.6	2541.8	2123.3	2209.5	2336.3	4293.5	**2006.4**
		t_{exp}	46.4	313.3	245.4	371.8	412.0	550.4	585.0	658.8	669.9	755.1	1047.5	**514.1**
		%mG	26.1	12.7	25.9	11.6	10.7	6.3	6.9	9.8	7.0	5.1	8.8	**11.9**
DSF	AGSP-RP	cpu_{dis}	0.2	0.2	0.3	0.1	0.2	0.2	0.2	0.2	0.3	0.3	0.5	**0.2**
		n	12.4	27.4	25.9	31.3	37.8	44.5	49.7	80.2	81.7	77.4	109.0	**52.5**
		cpu_{pre}	0.4	0.1	0.9	0.2	0.4	0.2	0.5	1.3	0.9	0.4	1.2	**0.6**
		cpu_{pla}	0.0	0.0	0.0	0.1	0.2	0.3	0.5	2.0	2.0	2.0	2.0	**0.8**
		cpu_{dpp}	0.5	0.3	1.2	0.4	0.8	0.6	1.2	3.4	3.3	2.6	3.7	**1.6**
		cpu_{rep}	0.0	0.2	0.3	0.4	1.2	2.5	4.3	44.7	51.2	35.2	96.0	**21.5**
		cpu_{dppr}	0.6	0.5	1.5	0.7	2.0	3.1	5.5	48.2	54.5	37.8	99.7	**23.1**
		t_{end}	772.8	1569.3	2173.8	1311.8	1586.3	2084.9	2487.6	2243.9	2323.0	2547.4	4466.8	**2142.5**
		t_{exp}	46.9	337.4	249.8	371.2	400.2	559.7	595.9	637.5	668.3	752.6	1029.2	**513.5**
		%mG	27.5	21.4	28.1	11.4	7.5	8.1	8.9	6.2	6.8	4.8	6.9	**12.5**

Table 4.11: Extended results for {DTF, WRF, DSF}+AGSP-RP. All times are in seconds.

Chapter 5

Final remarks

5.1 Conclusions

In this work, we propose an original solution to the *mobile search* problem. In general, the problem is solved by a standard decoupling approach; nevertheless, both parts of the solution are innovative and can also be seen as stand-alone contributions to related sub-problems — WRP and TDP. In our global solution, the main continuous problem is discretized by selecting a set of locations that completely cover the considered environment — *the discretization*. Then their visits' order is determined such that the expected time to find the *target* is minimized — *the routing*. About the same amount of work is dedicated to both solution stages.

In *the discretization*, we look for the shortest route covering the whole environment — a problem known in the literature as the WRP. The WRP is approached, again, by decomposition to two steps. First, a discrete set of covering points is generated. Second, the points' locations are optimized so that the shortest possible route connects them all while constrained that they remain to cover the whole environment. Our solution originally combines approaches from *computational geometry* and *combinatorial optimization*. The first step is achieved by customized *dual sampling* algorithm [13] utilizing solutions to modified MACS problem [14], and the optimization is done by combining solutions to the TSP [15], and TPP [16]. After obtaining the solution to the WRP, i.e., a route represented discretely as a sequence of covering locations connected by the shortest paths, the locations are de-sequenced and passed to the *mobile search* for re-sequencing that would minimize the second problem's optimization criterion. The overall idea behind

tackling the *mobile search discretization* obliquely by solving the WRP is based on the premise that particular discretizations well-fitted to either one of the problems would share some common properties. This hypothesis is confirmed only partially when tested on instances of the *mobile search*, as the proposed was not the best among tested methods, the others of which were taken directly from the literature. Our methods' strengths mostly show off around dense obstacle areas. However, as we found out, in *mobile search*, these areas are usually visited at later stages of the search and therefore have just a minor effect on the solution's quality. In wide-open areas that are the most desired to be searched efficiently and therefore crucial for good results, the method based on the dense *conforming constrained Delaunay triangulation* [11] exhibits better properties than our original method. Ultimately, we receive the best results from a hybrid that combines both approaches and a custom simple filtering that we additionally design. The filtering method efficiently reduces a set of locations to just those that are necessary for covering the whole environment. We show that filtered cover-locations yield better solutions to the *mobile search* in our experiments. In principle, the filtering method could also be used for solving the AGP by densely covering the environment and then applying the filtering, but this is not attempted in this work.

Regarding the *routing stage* of the *mobile search*, a significant portion of this thesis studies the simplest model — the TDP. We propose a new metaheuristic based on GVNS with restarts, deterministic VND, and custom double-bridge inspired *perturbations* for solving the TDP. Its design, together with selecting the best parametrization, is transparently described in a step-by-step fashion. The method's performance on the total of 120 benchmark instances of sizes 10-1000 is assessed by three types of experiments: (1) with hard upper limit on computational time, (2) with target solution cost, and (3) with fixed number of iterations. In the first two contexts (1, 2), the proposed method stably outperforms a reference metaheuristic suggested by Silva et al. [30]. Therefore, the proposed method is suitable for real-time applications, e.g., in mobile robotics, where the best possible solution is required while a hard limit bounds the time of computation. In the literature's most classical context (3), the proposed method provides solutions of better quality in exchange for longer running times for instances of size 500, while for smaller instances, this trend is observed reversely with the reference method. For the 500-customer instances, the proposed method found four new best known solutions.

As future improvements of the proposed metaheuristic for the TDP, we contemplate an extension to the MDM version similarly as the authors of [74] do with GILS-RVND. Application of DM techniques will not change the main results presented in this thesis. It will, however, further improve the performance of Ms-GVNS on larger instances and in scenarios without strict time constraints.

Further in the thesis, we extend the TDP to better model the *mobile search*. The first one of the extended models, GSP, considers weights associated with locations as a way to deal with non-equal locations' gains. The weights are proportional to the areas of *regions sensed* by the robot from the locations. The second extension, ATDP, considers an effort needed to turn the robot, which is modeled by combining two and three-dimensional costs. One another suggested *routing problem*, GSP2, models sensing on the way between the cover-locations by considering two-dimensional weights computed as the areas of regions perceived from the shortest routes between locations. Two more models — AGSP and AGSP2 — are created as the result of combining the ones previously mentioned. Four of the new models — ATDP, AGSP, GSP2, AGSP2 — are original to this work to the best of our knowledge. We adapt the proposed metaheuristic for the TDP to the new models by considering specialized total solution costs and separate efficient calculations of operator improvements. The derivations of the improvements are analogous to those for the TDP and are part of the projects' work too. Lastly, as the search progresses, we propose replanning as a simple trick to deal with the non-static character of locations' gains.

The developed metaheuristics and the proposed *discretization* methods are integrated into a software framework for the *mobile search*. Designing and implementing the framework is part of the work as well. A realistic simulation with a commercial robot, *TurtleBot3* [88], running on ROS, and a simple ideal stand-alone simulator are part of the framework. The ideal simulator was designed to perform large-scale experiments faster and allow replanning before starting a plan's execution. It assumes that the robot's movements are composed of simple maneuvers: going ahead with velocity v_{lin} and turning on the spot with velocity v_{ang}. The velocity constants are estimated based on the realistic simulations with *TurtleBot3*. In our experiments, we show that the two types of simulations provide, to some extent, comparable results.

Using the framework, we thoroughly evaluate 36 combinations of the *discretization* and *routing* variants on 11 *mobile search* instances and analyze the results. The solution method that provides the best average plans benefits from combining the hybrid *discretization* and AGSP model with replanning. The second best method uses the same *routing* but simpler triangulation-based *discretization*. However, it can provide the first plan in about 3-times lesser computational time than the best one. Lastly, we also confirm that our suggested model improvements indeed enhance the resulting plans, except for the two-dimensional weights. Some additional future work is required to adjust the weights' computing for the GSP2 and AGSP2 such that they stress out the beginnings of the shortest routes, which would be in line with the overall goal of the *mobile search*.

5.2 Publication plans

We intend to publish the key results created within this diploma project on an international forum. In this last section, we reveal our current publication plans.

This work's contributions can be divided into three major parts — TDP-related, WRP-related, and related to the *mobile search* as a whole — the first of which is complete, and the second two require slight amount of additional work. We intend to publish them separately. More specifically, as follows.

1. The proposed metaheuristic for the TDP, Ms-GVNS, has been already described in the article Mikula, J., Kulich, M. "A metaheuristic for the Traveling Deliveryman Problem in applications with a hard limit on computational time," which is currently in review of the journal *International Transactions in Operational Research* (IF 2.987).

2. The proposed method for *the discretization* based on covering the environment by MCCS, together with solution methods for the *traveling salesmen problem with neighborhoods* (TSPN), appears to provide good quality solutions to the WRP. The TSPN methods were not particularly discussed in this work, but we want to employ those introduced in [82] that utilize solutions to the TPP similarly to our proposed `Improve` method (Sec. 3.2.2), but in a more sophisticated way. We plan to present the overall combination in an article and send it for review to *IEEE Robotics and Automation Letters* (IF 3.608).

3. The complete solution to the *mobile search* will be presented in another article that we intend to send to *IEEE Transactions on Robotics* (IF 6.123). In the definitive paper, in addition to experiments performed in this thesis, we also want to compare our solution to Sarmiento et al. [3, 5] that introduce the only other complex solution methods for the *mobile search* known to us at this moment. We did not achieve the comparison in the thesis because Sarmiento et al. use slightly different assumptions about the problem, and therefore the transition of their solution methods to our framework is not straightforward. The authors in [3] assume differently from us the following: the robot is modeled as a single point, the sensor has unlimited range and sensing is performed only at the covering locations, i.e., not throughout the whole search independently of the robot's position, as in our case. Continuous sensing is assumed in [5], but the rest of the assumptions stays the same. Implementing and adapting their solution methods to our current framework requires some additional work that we will carry out in the future.

Appendix A

The TDP metaheuristic design

An extended methodology for comparing a lot of different metaheuristics on many instances of the problem is introduced in Sec. A.1. Then, the best metaheuristic design is done in two phases. First, several promising sets of neighborhoods are selected for consideration in the improvement procedure. The process is described in Sec. A.2. Second, many methods' configurations are tested, and the best is selected as the proposed method. Sec. A.3 provides the details. The final method's specification is done in the last Sec. A.4.

A.1 Methodology

While designing the best metaheuristic, on many occasions, we need to compare the performance of a lot of distinct variants on many different instances by some adequate metric. For this purpose, we extend the TTT-plots methodology presented in Sec. 1.3.5 by one extra step. We consider a reference algorithm $\mathcal{A}_R$ and z instances of the TDP labeled as $Inst = \{1, \ldots, z\}$. For each given instance $i \in Inst$, TTT-plots are computed for the reference and the tested method. Then, the probability $p^i_{tr} = P(RT^i_T < RT^i_R)$ is computed as described above. Here, RT^i_T and RT^i_R are random variables representing the time needed by the tested algorithm and the reference algorithm respectively to find a solution of an instance i which is as good as the given target value. Then, for every tested algorithm, we

report the final metrics

$$\textit{mean probability}: \ \text{mp} = \frac{\sum_i p_{tr}^i}{Z}, \text{ and} \tag{A.1}$$

$$\textit{weighted mean probability}: \ \text{wmp} = \frac{\sum_i size(i) \cdot p_{tr}^i}{\sum_i size(i)}, \tag{A.2}$$

where $size(i)$ is the number of vertices of the instance i. We choose GILS-RVND [30] as the reference algorithm for reasons explained in Sec. 3.3.1.

A.2 Promising neighborhoods

Here, all the general schemes are considered. Namely: GVNS, GRASP, and G+G, i.e., Alg. 3.8, 3.9, and 3.10 respectively. Also, two extra options were taken into account for G+G. The inner GVNS cycle inside G+G stops either after a fixed number of iterations j_{max} or after j_{max} iterations without improvement. The two versions of G+G are called G+G-(4a), and G+G-(4b) respectively with a reference to Point (4) of Sec. 1.1. The parameters of the algorithms are all fixed, with the exception of neighborhood structures. The parameters are chosen as follows: $\mathcal{P} = (4, 8, 12, 16)$, $\mathcal{R} = \{.00, .01, \ldots, .25\}$, and $j_{max} = 10$. The methods are tested on 15 standard benchmark instances TRP-S{50,100,200}-R{1,2,3,4,5} from Salehipour et al. [29]. For each pair consisting of a method and an instance, $n_{run} = 200$ runs are performed to compute a single TTT-plot. From each method's TTT-plots over all instances, the mp and wmp values are calculated as it is described in Sec. A.1. The target solution cost for each instance is chosen to be 1.01-multiple of the best solution reported by [30]. Also, in this set of experiments, RVND is preferred over the fixed-sequence VND. Thus, we do not take the order of neighborhoods into account, and the notion of sets instead of sequences is applicable. The way of selecting the most promising sets of neighborhoods is described next.

Assuming we have o_{max} operators available, then the total number of their combinations used in RVND is

$$\binom{o_{max}}{1} + \binom{o_{max}}{2} + \cdots + \binom{o_{max}}{o_{max}-1} + \binom{o_{max}}{o_{max}}. \tag{A.3}$$

For $o_{max} = 8$, we get $8+28+56+70+56+28+8+1 = 255$ available combinations. To reduce the number, we heuristically select just some of them. First, a set of all 8 neighborhoods $\boldsymbol{\mathcal{M}_8} = \{\mathcal{N}_{op} : op \in \boldsymbol{ops8}\}$, $\boldsymbol{ops8} = \{$`2-opt`, `1-point`, `or-opt2`, `or-opt3`, `or-opt4`, `or-opt5`, `2-point`, `3-point` $\}$, and its 8 corresponding subsets $\boldsymbol{\mathcal{M}_{8\setminus op}} = \boldsymbol{\mathcal{M}_8} \setminus \{\mathcal{N}_{op}\}$, each containing one lesser neighborhood than $\boldsymbol{\mathcal{M}_8}$, are tested as part of the 4 considered algorithms. The results are averaged over the algorithms and the best set among $\boldsymbol{\mathcal{M}_{8\setminus op}}$, $op \in \boldsymbol{ops8}$, is chosen and denoted

	Scheme	2-opt	1-point	or-opt2	or-opt3	or-opt4	or-opt5	2-point	3-point	%wmp	%mp
1	GVNS	✓	✓		✓					52.1	52.7
2	GVNS	✓	✓	✓		✓				50.1	51.1
3	GVNS	✓	✓		✓	✓				49.9	50.9
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
19	G+G-(4b)	✓	✓	✓		✓				46.0	46.6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
38	G+G-(4a)	✓	✓	✓		✓				42.9	44.3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
106	GRASP	✓	✓	✓		✓	✓	✓		15.7	22.4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
223	G+G-(4b)		✓							0.8	1.1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
226	GVNS		✓							0.2	0.3
227	G+G-(4a)		✓							0.2	0.3
228	GRASP	✓								0.1	0.2
										avg	avg
	GVNS	-	-	-	-	-	-	-	-	30.7	32.3
	G+G-(4b)	-	-	-	-	-	-	-	-	26.9	28.4
	G+G-(4a)	-	-	-	-	-	-	-	-	25.8	27.7
	GRASP	-	-	-	-	-	-	-	-	9.1	13.7

Table A.1: Selecting promising neighborhoods: full results. Ordered by the %wmp column, shortened.

as $\boldsymbol{\mathcal{M}_7}$. Then, the set of considered operators is changed to $\boldsymbol{ops7} \leftarrow \boldsymbol{ops8} \setminus (\boldsymbol{\mathcal{M}_8} - \boldsymbol{\mathcal{M}_7})$. In the next iteration, sets $\boldsymbol{\mathcal{M}_{7\setminus op}}$, $op \in \boldsymbol{ops7}$, are tested and evaluated analogously as above. The same is repeated for $\boldsymbol{\mathcal{M}_6}, \ldots, \boldsymbol{\mathcal{M}_1}$. We follow this strategy in a relaxed way, meaning that at some points, we add extra promising branches to the exploration tree. Eventually, we end up with 57 sets of neighborhoods tested in the four considered schemes. The results are recorded in a table and ordered by the value of wmp. The table has $57 \cdot 4 = 228$ rows, and its shortened version is shown in Tab. A.1. The shortened table shows the best three configurations on rows 1-3 and then the rows, where each algorithm appears for the first or for the last time. The symbol ✓ in the operator's column means that the operator was included in the tested neighborhood set. %mp and %wmp are the metrics described in Sec. A.1 in percents, rounded. The second metric, %wmp, emphasizes good performance on larger instances, which is convenient and we favor it over %mp.

	2-opt	1-point	or-opt2	or-opt3	or-opt4	or-opt5	2-point	3-point	%$\overline{\text{wmp}}$	%$\overline{\text{mp}}$
1	✓	✓	✓		✓				46.4	47.3
2	✓	✓		✓					46.1	47.8
3	✓	✓	✓		✓		✓		45.1	46.7
4	✓	✓	✓	✓	✓				44.9	45.3
5	✓	✓	✓	✓					44.8	45.9
6	✓	✓	✓	✓	✓		✓		44.6	45.5
7	✓	✓		✓	✓				44.4	45.7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
35	✓	✓							34.2	39.5
36	✓		✓	✓	✓		✓		10.4	13.8
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
46	✓		✓	✓	✓	✓		✓	6.0	8.1
47		✓	✓	✓	✓	✓	✓		5.9	8.1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
57		✓							0.4	0.6

Table A.2: Selecting promising neighborhoods: operator sets comparison. Ordered by %$\overline{\text{wmp}}$ column, shortened.

We can observe that the location of a particular configuration in the table is dependent both on the considered operators and the algorithm. The first 18 rows are ruled by GVNS only and the first 105 rows only by GVNS and G+Gs. The first appearance of pure GRASP is in row 106 with the value of %wmp 15.7, which is not much compared to the first appearance of GVNS, G+G-(4b), and G+G-(4a) with the values of %wmp 52.1, 46.0, and 42.9 respectively. To better illustrate how GRASP lacks behind the other methods, we show the averaged values over the operators at the bottom of Tab. A.1. Here we can see that the average value of %wmp is just 9.1 for GRASP while for the other methods it is over 25. Note, that the most significant difference between G+Gs and pure GRASP is that G+Gs contain the perturbation phase, which now seems to be crucial for TDP solving. This confirms the so-far leading role of GVNS, which contains the perturbation and not the randomized restarts. We might even hypothesize, that the lack of randomized restarts is what puts GVNS before G+Gs. Based on these interim results, we decide not to consider the pure GRASP algorithm anymore. Its performance is deficient compared to other methods, and its core idea, randomized restarts, can be represented as well by G+Gs in later phases of testing. The final results of this testing stage are shown in the shortened Tab. A.2. The full version of this table is obtained by removing all rows with GRASP from the full version of Tab. A.1, then averaging over the algorithms, and ordering the rows by the values of %$\overline{\text{wmp}}$. The best seven sets of operators are in rows

1-7 of Tab. A.2. These sets provide the best performance on average, throughout all the considered instances and algorithms. Note that they have some common features. For example, all of them contain operators `2-opt`, `1-point`, and at least one of `or-opt2`, `or-opt3`, `or-opt4`, and none of them contains `or-opt5`, or `3-point`. In addition, the special importance of `2-opt` and `1-point` can be seen from the table as a whole. Combinations, which contain both operators, are all in the first 35 rows. Note that the worst among the first 35, is the one where only `2-opt` and `1-point` are present and no other operator with them. The first row, which does not contain `1-point` is the 36th, and a significant jump (from 34.2 to 10.4) in the value of $\%\overline{\text{wmp}}$ can be seen when compared with the 35th row. All combinations which contain `2-opt` and not `1-point` are in the rows 36-46, and all combinations which contain `1-point` and not `2-opt` are in the rows 47-57. From these observations, we can conclude that `2-opt` on its own is better than `1-point` on its own, however, to get a significant boost in performance, both operators must be present in the considered set. To get even better results, 1-4 other operators can be taken into account, excluding `or-opt5`, and `3-point`. The most promising sets of operators that we select for further consideration are the best six (above the line) of Tab. A.2, and we entitle them as $\boldsymbol{\mathcal{N}_1^\star}, \ldots, \boldsymbol{\mathcal{N}_6^\star}$ respectively.

A.3 Finding the best variant

Here, three basic algorithms GVNS, G+G-(4a), and G+G-(4b) are considered together with the sets of promising neighborhoods $\boldsymbol{\mathcal{N}_1^\star}, \ldots, \boldsymbol{\mathcal{N}_6^\star}$. In addition to RVND - the improving procedure used so far - we also consider its original fixed-sequence version - VND. For the VND, a specific order of the neighborhood structures must be taken into account. We use the same order as in which the operators appear in Tab. A.2. This choice is motivated by the importance of `2-opt`, and `1-point`, which are put on the first and the second place in the sequence respectively. The rest of the operators are put in the sequence arbitrarily ordered, as no significant observations of their effect on the performance are made. Furthermore, four different configurations of the perturbation parametrization are considered: $\mathcal{P}_1 = (4)$, $\mathcal{P}_2 = (4, 8)$, $\mathcal{P}_3 = (4, 8, 12)$, and $\mathcal{P}_4 = (4, 8, 12, 16)$. The constructive heuristic in G+Gs has several different variants as well. All variants are implemented by `Construct`(s_{rcl}) procedure as it is discussed in Sec 3.3.4, where $s_{rcl} = 1$ for the deterministic strategy, $s_{rcl} = 3$ for the fixed-randomness strategy, and s_{rcl} is constructed as in line 5 of Alg. 3.9 with $\mathcal{R} = \{.00, .01, \ldots, .25\}$ for the variable-randomness strategy. Let us call these construction strategies det, fixed-s_{rcl} and rand-s_{rcl} respectively. Also, for both G+G heuristics, 8 different values of the inner iteration constant j_{max} are considered: $j_{max} \in \{10, 20, 30, 40, 50, 100, 150, 200\}$.

	Scheme	Constr.	j_{max}	RVND	$\mathcal{N}^{\star}_{(\cdot)}$	$\mathcal{P}_{(\cdot)}$	%wmp	%mp
1	G+G-(4b)	det	30		4	3	72.2	74.3
2	G+G-(4a)	det	40		5	3	72.1	74.1
3	G+G-(4a)	det	40		4	4	71.9	74.7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
38	GVNS				5	3	71.0	73.3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
56	G+G-(4b)	det	50		5	2	70.8	72.4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
274	G+G-(4b)	fixed-s_{rcl}	20		4	4	66.6	69.9
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
276	G+G-(4b)	det	200		2	1	66.4	68.6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
287	G+G-(4b)	rand-s_{rcl}	10		4	4	66.0	68.9
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
572	G+G-(4b)	det	50	✓	2	2	63.5	66.6
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2352	G+G-(4a)	rand-s_{rcl}	10		2	1	30.1	45.0

Table A.3: Finding the best metaheuristic: full results. Ordered by the %wmp column, shortened.

Ultimately, we consider two versions of the improvement procedure, six sets / sequences of neighborhood structures, and four different perturbation parametrizations for the GVNS, i.e., $2 \cdot 6 \cdot 4 = 48$ different configurations. For G+Gs, we have in addition: three versions of the construction procedure, and eight values for the J parameter, therefore $48 \cdot 3 \cdot 8 = 1152$ different configurations. Overall, $48 + 2 \cdot 1152 = 2352$ heuristics' configurations are tested on 15 instances TRP-S{50,100,200}-R{1,2,3,4,5} [29], and $n_{run} = 200$ runs for each method-instance pair are performed. The produced run-times are processed in accordance with the methodology of Sec. A.1. The results in a form of mp and wmp metrics for each method configuration are recorded into a 2352-row table, whose shortened version we show in Tab. A.3. The symbol ✓ in RVND column means RVND is used as the improvement procedure. The absence of ✓ means VND is used. The number $\boldsymbol{q} \in \{1, \ldots, 6\}$ in column $\mathcal{N}^{\star}_{(\cdot)}$ means $\mathcal{N}^{\star}_{\boldsymbol{q}}$ is used as the neighborhood set/sequence, and the number $g \in \{1, \ldots, 4\}$ in column $\mathcal{P}_{(\cdot)}$ means the sequence $\mathcal{P}_g$ is used as the perturbation parametrization. Values %wmp and %mp are the metrics described in Sec. A.1, in percents, rounded.

	Construction	RVND	$\mathcal{P}_{(\cdot)}$	%$\overline{\text{wmp}}$	%$\overline{\text{mp}}$
1	det		4	69.7	73.0
2	det		3	69.2	72.5
3	det		2	65.9	69.9
4	fixed-s_{rcl}		4	63.6	68.0
5	rand-s_{rcl}		4	63.1	67.7
6	fixed-s_{rcl}		3	62.2	66.8
7	rand-s_{rcl}		3	61.7	66.6
8	det	✓	2	59.6	63.2
9	det	✓	3	58.8	62.8
10	fixed-s_{rcl}		2	57.2	63.2
11	det	✓	4	57.1	61.6
12	rand-s_{rcl}		2	56.7	62.9
13	det	✓	1	56.3	60.8
14	det		1	54.8	61.3
15	fixed-s_{rcl}	✓	2	54.6	58.7
16	fixed-s_{rcl}	✓	3	54.2	58.4
17	rand-s_{rcl}	✓	2	53.6	57.8
18	rand-s_{rcl}	✓	3	53.5	57.8
19	fixed-s_{rcl}	✓	4	52.9	57.6
20	rand-s_{rcl}	✓	4	52.1	56.9
21	fixed-s_{rcl}	✓	1	50.2	55.4
22	rand-s_{rcl}	✓	1	49.0	54.6
23	fixed-s_{rcl}		1	44.2	53.4
24	rand-s_{rcl}		1	43.3	52.9

Table A.4: Comparison of construct-improve-perturbate strategies for G+G-(4a). Ordered by the %$\overline{\text{wmp}}$ column.

The best three heuristics' configurations, the worst one, and some notable others, are displayed in Tab. A.3. The notable configurations include the following rows: 38, where GVNS firstly appears; 56, where $\mathcal{P}_2$ firstly appears; 274, where fixed-s_{rcl} firstly appears; 276, where $\mathcal{P}_1$ firstly appears; 287, where rand-s_{rcl} firstly appears; and 572, where RVND firstly appears. Let us, for the rest of this paragraph, discuss how certain components of certain configurations can have an effect on the value of wmp. Note, that the configurations in the head of the table carry some common features. They follow one of G+G schemes, they are fully deterministic except the perturbation, and the perturbation parametrization is either $\mathcal{P}_3$ or $\mathcal{P}_4$. To better illustrate how different construct-improve-perturbate strategies effect the resulting %wmp, we show Tab. A.4. Here, we select the rows of a single scheme - G+G-(4a) and average them over j_{max}, and $\boldsymbol{\mathcal{N}}^{\star}_{(\cdot)}$ columns. It is clear from the table, that the strategies with the highest value of %$\overline{\text{wmp}}$ (rows 1-3) use deterministic construction, VND and $\mathcal{P}_4$, $\mathcal{P}_3$, and $\mathcal{P}_2$. The next best-performing methods (rows 4-12) use $\mathcal{P}_4$, $\mathcal{P}_3$, or $\mathcal{P}_2$ and only one of these two features: deterministic construction, and VND. The rest of the methods (13-24) use

either both randomized construction and RVND, or $\mathcal{P}_1$. On average, deterministic construction is better then randomized, s_{rcl}-fixed construction is better than s_{rcl}-rand construction, and VND is better than RVND. If the construction and the improvement procedure are fixed, then using $\mathcal{P}_1$ in perturbation is always the worst option. If VND is used, then the higher q for $\mathcal{P}_q$, the higher value of $\%\overline{\text{wmp}}$. If RVND is used, then the best option for q is 2. Similar conclusions can be made for the other schemes as well.

A.4 The final method

According to Tab. A.3, the best overall method configuration is the G+G-(4b) with deterministic construction, $j_{max} = 30$, VND, $\boldsymbol{\mathcal{N}} = (\mathcal{N}_{\texttt{2-opt}}, \mathcal{N}_{\texttt{1-point}}, \mathcal{N}_{\texttt{or-opt2}}, \mathcal{N}_{\texttt{or-opt3}}, \mathcal{N}_{\texttt{or-opt4}})$, and $\mathcal{P} = (4, 8, 12)$. The best heuristic's scheme is based on G+G general scheme, however, the construction it uses is deterministic, and therefore the relation with GRASP is no longer accurate. Let us denote the metaheuristic as *multi-start GVNS* (Ms-GVNS). Our final observation regarding Ms-GVNS is that for larger instances (including those of size 1000) it works better with higher j_{max}. Therefore, the final method is the same as described above, except $j_{max} = \lceil size(i)/5 \rceil$ instead of being fixed to 30. Here, $size(i)$ is the size of the instance i being solved.

Appendix B

List of abbreviations

Abbr.	Meaning
AGP	*art gallery problem*
AGSP	GSP extended by considering three-dimensional costs
AGSP-RP	AGSP with replanning
AGSP2	AGSP modified by considering two-dimensional weights
AGSP2-RP	AGSP2 with replanning
ATDP	TDP extended by considering three-dimensional costs
CDF	*cumulative distribution function*
DM	*data mining*
DS	*discretization* based on *dual sampling* algorithm
DSF	DS with filtering
DT	*discretization* based on *conforming constrained Delaunay triangulation*
DTF	DT with filtering
FIM	*frequent itemset mining*
G+G	GRASP + GVNS
GILS	GRASP-ILS
GRA	*greedy randomized adaptive* (adj.)
GRASP	*GRA search procedure*
GSP	*graph search problem*
GSP-RP	GSP with replanning
GSP2	GSP modified by considering two-dimensional weights
GVNS	*generalized VNS*
IF	*impact factor*
ILP	*integer linear programming*
ILS	*iterated local search*
KA	*discretization* based on Kazazakis and Argyros
KAF	KA with filtering
LK	*Lin-Kernighan heuristic*

LKH	Helsgaun's effective implementation of the LK
MACS	*maximum area convex subset*
MCCS	*maximally covering convex subset*
MDM	*multi-DM*
MILP	*mixed integer linear programming*
mp	*mean probability*
Ms-GVNS	*multi-start GVNS*
PDF	*probability density function*
ROS	*Robot Operating System*
RP	replanning
RTD	*run-time distribution*
RVND	*randomized VND*
TPP	*touring polygons problem*
TPPO	*TPP with obstacles*
TS	*tabu search*
TSP	*traveling salesman problem*
TSPN	*TSP with neighborhoods*
TTT	time to target
UB	upper bound
VND	*variable neighborhood descent*
VNS	*variable neighborhood search*
VRP	*vehicle routing problem*
wmp	*weighted mean probability*
WR	*discretization* based on WRP
WRF	WR with filtering
WRF-DT-F	*discretization* combining WRF and DT, with filtering
WRP	*watchman route problem*

Appendix C

Bibliography

[1] C. Robin and S. Lacroix, "Multi-robot target detection and tracking: taxonomy and survey," *Autonomous Robots*, vol. 40, pp. 729–760, apr 2016.

[2] A. Sarmiento, R. Murrieta, and S. Hutchinson, "An efficient strategy for rapidly finding an object in a polygonal world," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, vol. 2, pp. 1153–1158, IEEE, 2003.

[3] A. Sarmiento, R. Marrieta-Cid, and S. Hutchinson, "Planning expected-time optimal paths for searching known environments," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 1, pp. 872–878, IEEE, 2004.

[4] A. Sarmiento, R. Murrieta-Cid, and S. Hutchinson, "A multi-robot strategy for rapidly searching a polygonal environment," in *Advances in Artificial Intelligence – IBERAMIA 2004*, pp. 484–493, Springer Berlin Heidelberg, 2004.

[5] A. Sarmiento, R. Murrieta-Cid, and S. Hutchinson, "An Efficient Motion Strategy to Compute Expected-Time Locally Optimal Continuous Search Paths in Known Environments," *Advanced Robotics*, vol. 23, pp. 1533–1560, jan 2009.

[6] J. Lv, M. Liu, H. Zhao, B. Li, and S. Sun, "Maritime Static Target Search Based on Particle Swarm Algorithm," in *SAI Intelligent Systems Conference*, pp. 917–927, 2018.

[7] M. Kulich, L. Přeučil, and J. J. M. Bront, "Single robot search for a stationary object in an unknown environment," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5830–5835, IEEE, may 2014.

[8] M. Kulich, J. J. Miranda-Bront, and L. Přeučil, "A meta-heuristic based goal-selection strategy for mobile robot search in an unknown environment," *Computers & Operations Research*, vol. 84, pp. 178–187, aug 2017.

[9] M. Kulich, L. Přeučil, and J. J. Miranda Bront, "On multi-robot search for a stationary object," in *2017 European Conference on Mobile Robots (ECMR)*, pp. 1–6, IEEE, sep 2017.

[10] M. Kulich and L. Přeučil, "Multirobot search for a stationary object placed in a known environment with a combination of GRASP and VND," *International Transactions in Operational Research*, apr 2020.

[11] J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry*, vol. 22, pp. 21–74, may 2002.

[12] G. Kazazakis and A. Argyros, "Fast positioning of limited-visibility guards for the inspection of 2D workspaces," in *IEEE/RSJ International Conference on Intelligent Robots and System*, vol. 3, pp. 2843–2848, IEEE, 2002.

[13] H. H. González-Baños, , and J.-C. Latombe, *Planning robot motions for range-image acquisition and automatic 3d model construction*. Citeseer, 1998.

[14] D. Coeurjolly and J. Chassery, "Fast approximation of the maximum area convex subset for star-shaped polygons," *RR-LIRIS-2004-006*, 2004.

[15] W. J. Cook, *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.

[16] X. Pan, F. Li, and R. Klette, "Approximate shortest path algorithms for sequences of pairwise disjoint simple polygons," *Proceedings of the 22nd Annual Canadian Conference on Computational Geometry, CCCG 2010*, pp. 175–178, 2010.

[17] S. Sahni and T. Gonzalez, "P-Complete Approximation Problems," *Journal of the ACM (JACM)*, vol. 23, pp. 555–565, jul 1976.

[18] E. Koutsoupias, C. Papadimitriou, and M. Yannakakis, "Searching a fixed graph," in *Automata, Languages and Programming*, pp. 280–289, Springer Berlin Heidelberg, 1996.

[19] A. Archer and D. P. Williamson, "Faster approximation algorithms for the minimum latency problem," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 88–96, SIAM, 2003.

[20] M. Fischetti, G. Laporte, and S. Martello, "The Delivery Man Problem and Cumulative Matroids," *Operations Research*, vol. 41, pp. 1055–1064, dec 1993.

[21] G. Ausiello, S. Leonardi, and A. Marchetti-Spaccamela, "On salesmen, repairmen, spiders, and other traveling agents," in *Algorithms and Complexity*, pp. 1–16, Springer Berlin Heidelberg, 2000.

[22] A. M. Campbell, D. Vandenbussche, and W. Hermann, "Routing for Relief Efforts," *Transportation Science*, vol. 42, pp. 127–145, may 2008.

[23] J. Faigl and G. A. Hollinger, "Unifying multi-goal path planning for autonomous data collection," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2937–2942, IEEE, sep 2014.

[24] I. Gentilini, F. Margot, and K. Shimada, "The travelling salesman problem with neighbourhoods: MINLP solution," *Optimization Methods and Software*, vol. 28, pp. 364–378, apr 2013.

[25] S. L. Smith and F. Imeson, "GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem," *Computers and Operations Research*, vol. 87, pp. 1–19, 2017.

[26] A. Gunawan, H. C. Lau, and P. Vansteenwegen, "Orienteering Problem: A survey of recent variants, solution approaches and applications," *European Journal of Operational Research*, vol. 255, pp. 315–332, dec 2016.

[27] H. Abeledo, R. Fukasawa, A. Pessoa, and E. Uchoa, "The time dependent traveling salesman problem: polyhedra and algorithm," *Mathematical Programming Computation*, vol. 5, pp. 27–55, mar 2013.

[28] L. M. Naeni and A. Salehipour, "A New Mathematical Model for the Traveling Repairman Problem," in *2019 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pp. 1384–1387, IEEE, dec 2019.

[29] A. Salehipour, K. Sörensen, P. Goos, and O. Bräysy, "Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem," *4OR*, vol. 9, pp. 189–209, 2011.

[30] 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, sep 2012.

[31] N. Mladenović, D. Urošević, and S. Hanafi, "Variable neighborhood search for the travelling deliveryman problem," *4OR*, vol. 11, no. 1, pp. 57–73, 2013.

[32] T. A. Feo, M. G. C. Resende, and S. H. Smith, "A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set," *Operations Research*, vol. 42, pp. 860–878, oct 1994.

[33] J. Mikula, "Meta-heuristics for routing problems," Bachelor's thesis, CTU in Prague, 2018.

[34] J. O'Rourke, *Art Gallery Theorems and Algorithms*. USA: Oxford University Press, Inc., jun 1987.

[35] V. Chvátal, "A combinatorial theorem in plane geometry," *Journal of Combinatorial Theory, Series B*, vol. 18, pp. 39–41, feb 1975.

[36] W.-p. Chin and S. Ntafos, "Optimum watchman routes," *Information Processing Letters*, vol. 28, pp. 39–44, may 1988.

[37] J. Faigl, *Multi-goal path planning in mobile robotic tasks*. Habilitation, CTU in Prague, 2013.

[38] K. A. Eldrandaly, A. N. Ahmed, and A. F. AbdAllah, "Routing Problems : A Survey," in *The 43rd Annual Conference on Statistics, Computer Sciences and Operations Research*, 2008.

[39] A. Mor and M. G. Speranza, "Vehicle routing problems over time: a survey," *4OR*, vol. 18, pp. 129–149, jun 2020.

[40] F. W. Glover and G. A. Kochenberger, eds., *Handbook of metaheuristics*. New York, NY: Springer, 2003 ed., 2006.

[41] Wikipedia, "Metaheuristic — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Metaheuristic&oldid=993135619`, 2020. [Online; accessed 12-December-2020].

[42] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, pp. 1097–1100, nov 1997.

[43] J. P. Hart and A. W. Shogan, "Semi-greedy heuristics: An empirical study," *Operations Research Letters*, vol. 6, pp. 107–114, jul 1987.

[44] T. A. Feo and M. G. C. Resende, "A probabilistic heuristic for a computationally difficult set covering problem," *Operations Research Letters*, vol. 8, pp. 67–71, apr 1989.

[45] H. H. Hoos and T. Stützle, "Evaluating Las Vegas Algorithms - Pitfalls and Remedies," *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 238–245, jan 1998.

[46] R. M. Aiex, M. G. Resende, and C. C. Ribeiro, "Probability distribution of solution time in GRASP: An experimental investigation," *Journal of Heuristics*, vol. 8, no. 3, pp. 343–373, 2002.

[47] M. G. C. Resende and C. C. Ribeiro, *Optimization by GRASP*. New York, NY: Springer New York, 2016.

[48] C. C. Ribeiro, I. Rosseti, and R. Vallejos, "On the use of run time distributions to evaluate and compare stochastic local search algorithms," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009.

[49] C. C. Ribeiro and I. Rosseti, "tttplots-compare: a perl program to compare time-to-target plots or general runtime distributions of randomized algorithms," *Optimization Letters*, vol. 9, pp. 601–614, mar 2015.

[50] C. C. Ribeiro, I. Rosseti, and R. Vallejos, "Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms," in *Journal of Global Optimization*, 2012.

[51] A. Lucena, "Time-dependent traveling salesman problem – the deliveryman case," *Networks*, vol. 20, no. 6, pp. 753–763, 1990.

[52] L. Bianco, A. Mingozzi, and S. Ricciardelli, "The traveling salesman problem with cumulative costs," *Networks*, vol. 23, pp. 81–91, mar 1993.

[53] C. A. van Eijl, "A polyhedral approach to the delivery man problem," Tech. Rep. 1995, Technische Universiteit Eindhoven (Memorandum COSOR), Eindhoven, 1995.

[54] I. Méndez-Díaz, P. Zabala, and A. Lucena, "A new formulation for the Traveling Deliveryman Problem," *Discrete Applied Mathematics*, vol. 156, pp. 3223–3237, oct 2008.

[55] I. O. E. Ezzine, H. Chabchoub, and F. Semet, "New formulations for the traveling repairman problem," in *Proceedings of the 8-th International Conference of Modeling and Simulation*, (Hammamet, Tunisia), MOSIM, 2010.

[56] H.-B. Ban, K. Nguyen, M.-C. Ngo, and D.-N. Nguyen, "An efficient exact algorithm for the Minimum Latency Problem," *Progress in Informatics*, p. 167, mar 2013.

[57] L. Gouveia and S. Voß, "A classification of formulations for the (time-dependent) traveling salesman problem," *European Journal of Operational Research*, vol. 83, pp. 69–82, may 1995.

[58] L.-P. Bigras, M. Gamache, and G. Savard, "The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times," *Discrete Optimization*, vol. 5, pp. 685–699, nov 2008.

[59] H. Abeledo, R. Fukasawa, A. Pessoa, and E. Uchoa, "The time dependent traveling salesman problem: Polyhedra and branch-cut-and-price algorithm," in *Experimental Algorithms* (P. Festa, ed.), (Berlin, Heidelberg), pp. 202–213, Springer Berlin Heidelberg, 2010.

[60] J. J. Miranda-Bront, I. Méndez-Díaz, and P. Zabala, "An integer programming approach for the time-dependent TSP," *Electronic Notes in Discrete Mathematics*, vol. 36, pp. 351–358, aug 2010.

[61] J. J. Miranda-Bront, I. Méndez-Díaz, and P. Zabala, "Facets and valid inequalities for the time-dependent travelling salesman problem," *European Journal of Operational Research*, vol. 236, pp. 891–902, aug 2014.

[62] M. T. Godinho, L. Gouveia, and P. Pesneau, "Natural and extended formulations for the Time-Dependent Traveling Salesman Problem," *Discrete Applied Mathematics*, vol. 164, pp. 138–153, feb 2014.

[63] G. Reinelt, "TSPLIB - A Traveling Salesman Problem Library," *ORSA Journal on Computing*, vol. 3, pp. 376–384, nov 1991.

[64] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan, "The minimum latency problem," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing - STOC '94*, (New York, New York, USA), pp. 163–171, ACM Press, 1994.

[65] M. Goemans and J. Kleinberg, "An improved approximation ratio for the minimum latency problem," *Mathematical Programming, Series B*, 1998.

[66] S. Arora and G. Karakostas, "Approximation schemes for minimum latency problems," *SIAM Journal on Computing*, vol. 32, pp. 1317–1337, jan 2003.

[67] 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.*, pp. 36–45, IEEE Computer. Soc, 2003.

[68] J. Fakcharoenphol, C. Harrelson, and S. Rao, "The k -traveling repairmen problem," *ACM Transactions on Algorithms*, vol. 3, pp. 40–es, nov 2007.

[69] V. Nagarajan and R. Ravi, "The Directed Minimum Latency Problem," in *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pp. 193–206, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[70] A. Archer, A. Levin, and D. P. Williamson, "A Faster, Better Approximation Algorithm for the Minimum Latency Problem," *SIAM Journal on Computing*, vol. 37, pp. 1472–1498, jan 2008.

[71] A. Archer and A. Blasiak, "Improved Approximation Algorithms for the Minimum Latency Problem via Prize-Collecting Strolls," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, (Philadelphia, PA), pp. 429–447, Society for Industrial and Applied Mathematics, jan 2010.

[72] G. N. Frederickson and B. Wittman, "Approximation Algorithms for the Traveling Repairman and Speeding Deliveryman Problems," *Algorithmica*, vol. 62, pp. 1198–1221, apr 2012.

[73] E. F. S. Rios, *Exploração de Estratégias de Busca Local em Ambientes CPU/GPU*. PhD thesis, Universidade Federal Fluminense, 2016.

[74] Í. Santana, A. Plastino, and I. Rosseti, "Improving a state-of-the-art heuristic for the minimum latency problem with data mining," *International Transactions in Operational Research*, jan 2020.

[75] J.-B. Saloman, "Probability and Statistics (Fourth Edition)," *CHANCE*, vol. 26, pp. 54–54, sep 2013.

[76] P.-S. Laplace, *Théorie analytique des probabilités*. Paris: Courcier, 1812.

[77] C. I. Lewis and J. M. Keynes, "A Treatise on Probability.," *The Philosophical Review*, vol. 31, p. 180, mar 1922.

[78] X. Chen and S. McMains, "Polygon Offsetting by Computing Winding Numbers," in *Volume 2: 31st Design Automation Conference, Parts A and B*, pp. 565–575, ASMEDC, jan 2005.

[79] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Berlin, Heidelberg: Springer-Verlag, 1985.

[80] K. Helsgaun, "An effective implementation of the Lin–Kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, pp. 106–130, oct 2000.

[81] K. Helsgaun, "An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems," tech. rep., Roskilde University, 2017.

[82] J. Vidašič, "Travelling Salesman Problem with Neighborhoods," Master's thesis, CTU in Prague, 2020.

[83] O. Martin, S. W. Otto, and E. W. Felten, "Large-step Markov Chains for the Traveling Salesman Problem," *Complex Systems*, vol. 5, pp. 219–224, 1991.

[84] A. Mjirda, R. Todosijević, S. Hanafi, P. Hansen, and N. Mladenović, "Sequential variable neighborhood descent variants: an empirical study on the traveling salesman problem," *International Transactions in Operational Research*, 2017.

[85] D. Satyananda and S. Wahyuningsih, "Sequential order vs random order in operators of variable neighborhood descent method," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 17, no. 2, pp. 801–808, 2019.

[86] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, vol. 3, 2009.

[87] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

[88] R. Amsters and P. Slaets, "Turtlebot 3 as a robotics education platform," in *Robotics in Education* (M. Merdan, W. Lepuschitz, G. Koppensteiner, R. Balogh, and D. Obdržálek, eds.), (Cham), pp. 170–181, Springer International Publishing, 2020.

[89] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, pp. 23–33, mar 1997.

Appendix D

CD content

File	Description
`mlp.zip`	An archive containing `C++` source files of the solver for MLPs where the Ms-GVNS method is implemented.
`ros_ws.zip`	An archive containing ROS workspace with `C++` source files for the *mobile search* solution in the form of individual packages.
`results.zip`	An archive containing results of this thesis, i.e., some `CSV` files, LaTeX tables and `Python` scripts.