UNIVERSITY OF CALIFORNIA, MERCED

Planning Algorithms for Robots Operating in Vineyards

A dissertation presented for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Thomas C. Thayer

2021

Submitted by:	Thomas C. Thayer
Advisor:	Stefano Carpin, Ph.D. Electrical Engineering and Computer Science
Committee Members:	Marcelo Kallmann, Ph.D. Electrical Engineering and Computer Science
	Joshua Viers, Ph.D. Environmental Systems
	Stefano Carpin, Ph.D. (chair) Electrical Engineering and Computer Science

This dissertation is approved and accepted for publication by the committee.

Marcelo Kallmann

Joshua Viers

Stefano Carpin

Abstract

Planning Algorithms for Robots Operating in Vineyards
A dissertation presented for the degree of Doctor of Philosophy in
Electrical Engineering and Computer Science
By: Thomas C. Thayer
Committee chair: Stefano Carpin
University of California, Merced 2021

Contemporary vineyard management is in dire need of a way to make remote sensing data useful for irrigation purposes on the fine-grain scale. A robot can be used to adjust irrigation emitters within a vineyard, but first requires solving a difficult optimization problem, where a path must be planned that maximizes the cumulative adjustment of water emitters while the path's total length is limited by the battery life of the robot. This is formally called the Orienteering Problem, which is NP-hard. The physical structure of a vineyard constrains movement within it, and the colossal size of some vineyards means that special algorithms are needed to compute efficient solutions. Furthermore, useful extensions to this problem provide additional benefit for real-world vineyards. Solutions to the Team Orienteering Problem, which requires coordinated paths built for multiple agents, can make irrigation management more efficient using a team of robots. The Bi-Objective Orienteering Problem provides robot paths that can perform an additional task while adjusting water emitters, such as soil sampling for moisture content. These problems all assume deterministic movement costs, whereas robots traversing agricultural settings can encounter less desirable field conditions which reduce their speed. The Stochastic Orienteering Problem with Chance Constraints can be used to account for this uncertainty in path planning and provide a bound on the chance of failure. This problem is difficult to solve in large instances such as those encountered with agricultural routing, and therefore ways to speed up computation are needed.

This dissertation addresses each of these variations of the Orienteering Problem within the context of vineyards. A special type of model is discussed called an Aisle Graph which represents the structure of a typical vineyard. For the mentioned variations of the Orienteering Problem, heuristic path planners are created which take advantage of this unique arrangement to provide highly economical paths for robots very quickly. Each of the heuristic planners is analyzed on real-world problem simulations constructed with datasets from commercial vineyards located in central California, and shown to be efficient at directing robots to adjust irrigation emitters in fields containing tens of thousands of vines.

Related Publications

[121] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Routing algorithms for robot assisted precision irrigation. In *IEEE International Conference on Robotics and Automation*, pages 2221–2228, 2018

[120] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Multi-robot routing algorithms for robots operating in vineyards. In *IEEE International Conference on Automation Science and Engineering*, pages 14–21, 2018. Best Paper Award

[122] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Bi-objective routing for robotic irrigation and sampling in vineyards. In *IEEE International Conference on Automation Science and Engineering*, pages 1481–1488, 2019

[123] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Multirobot routing algorithms for robots operating in vineyards. *IEEE Transactions on Automation Science and Engineering*, 17(3):1184–1194, 2020

[116] Thomas C. Thayer and Stefano Carpin. Solving large scale stochastic orienteering problems with aggregation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2452–2458, 2020

[117] Thomas C. Thayer and Stefano Carpin. An adaptive method for the stochastic orienteering problem. *IEEE Robotics and Automation Letters*, 6(2):4185–4192, 2021

[118] Thomas C. Thayer and Stefano Carpin. A fast algorithm for stochastic orienteering with chance constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2021. (To Appear)

[119] Thomas C. Thayer and Stefano Carpin. A resolution adaptive algorithm for the stochastic orienteering problem with chance constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2021. (To Appear)

[72] Xinyue Kan, Thomas C. Thayer, Stefano Carpin, and Konstantinos Karydis. Task planning on stochastic aisle graphs for precision agriculture. *IEEE Robotics and Automation Letters*, 6(2):3287–3294, 2021

Acknowledgments

First, I would like to thank Stefano Carpin for his continued support and advice as my PhD advisor and the opportunity to have worked in his laboratory as a graduate and undergraduate student. Additionally, I would like to thank the other members of my advisatory committee, Marcelo Kallmann and Joshua Viers, for reviewing my dissertation. I would also like to thank the co-authors of each of my research papers and conference publications which became the foundation of my research.

Second, I would also like to thank the UC Merced community which I have been a part of for the last ten years. Having started as a naive freshman and working my way up to the rank of PhD student (including at one time staff member), there are many people with whom I have had the honor of calling colleague, co-worker, manager, lab-mate, or friend, and each of them deserves recognition for the part they played in my academic, professional, and personal development.

Finally, I would like to thank my family for their unwavering support through my entire life. My mother, father, sister, and every other family member have always been there for me, and I am forever grateful to have them as a part of my life. I know they always wanted a professional baseball player in the family, however I hope a doctor will suffice instead.

I also must acknowledge financial support by the USDA-NIFA under award number 2017-67021-25925 and the NSF under grant DGE-1633722.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Purpose

Agriculture is arguably one of the most important human activities, as it is used to grow food and other products for people to consume. Through the ages agriculture has evolved slowly compared to other industries, and this is true even in contemporary times. Some farms are still using centuries-old techniques that are costly and inefficient, however there is no better alternative or the alternative is too expensive. Luckily, this is changing. Agriculture technology (colloquially known as AgTech) has gained new interest from researchers at universities and private companies as the demand for agricultural products continues to increase toward the industry's capacity. New devices, techniques, and systems are put to the test and implemented in commercial operations frequently, creating new paradigms that are reaching even the humblest of farmers.

Water use efficiency in agricultural purposes is becoming a very popular topic to study. Droughts are a frequent occurrence in many parts of the world and can have detrimental long-term effects on the yield and quality of crop output in the effected areas, as well as inducing wider economic and civil issues. The ability to utilize water more efficiently means that drought years are less devastating and wet years can provide more water to replenish aquifers or dams, further decreasing the severity of arid years. In the United States of America, the majority of freshwater used goes to the agriculture industry, taking roughly 85% of the consumed supply[111].

The potential need for freshwater savings is particularly dramatic when examining multi-year crops. Since perennial plants are more complex in their growth patterns and nutrient requirements, it is difficult for farmers to precisely estimate the needs of their fields, and so they tend to error on the side of caution to keep their plants healthy. In terms of water, this means that growers will intentionally over-irrigate, wasting precious resources in the hopes of keeping their operations sustainable. This is especially true with some fruit orchards, where lack of watering in earlier years has been shown to negatively effect the yields of future years [69]. In California, where perennial crops and products account for over half ($11.8 billion in 2019) of agricultural exports [1], over-watering can be especially burdensome. And because these crops are a significant portion of California's economy, a prolonged drought or change in water resource policy can have drastic consequences for the state. Thus, there is a pressing necessity for developing ways to boost the efficiency of farmland irrigation.

Special consideration is required when investigating the irrigation needs of wine grapes. The eventual pressing and fermenting into wine means that these grapes need to meet stringent specifications to ensure quality in the final product. Fruit size and water content have a large impact on a number of these specifications, including brix and flavonoid content, meaning that high moisture has a direct negative effect on the character of the wine [75]. Because of this, vineyard managers need a special watering scheme that provides the optimum amount of water to their grapevines, preventing over-saturation of the fruits but also keeping the vines healthy. This watering scheme, called *deficit irrigation*, emphasizes slightly under-watering plants, potentially sacrificing plant vigor and overall yield but producing more economically desirable grapes [35]. Still, the conceivable threat of harming or killing a vine by accidentally over-stressing it with too little water, causing long-term economic loss, prevents ranch managers from attempting to employ the optimal irrigation schedule. Consequentially, vineyards are over-irrigated and quality is decreased.

There are two main factors that limit a farmer's potential for properly and efficiently irrigating their vineyards. These are sensing the needs of the vines, and delivering a precise amount of water. Generally, these factors are gauged at the block level, an area that may be as modest as a few acres for small private vineyards, to many hundreds of acres for the largest commercial wineries. Sensing plant water stress is done indirectly by measuring local atmospheric conditions to calculate evapotranspiration and directly using stem water potential or leaf water potential [39]. These methods suffer from lack of scalability; local atmospheric conditions generalize plant stress over many miles, and pressure chambers and bagging give readings useful for specific plants only. Ideally, plant stress should be established on a per-plant basis over the entire vineyard, meaning data should be both highly specific and cover a large area. Some methods have emerged to do this, usually involving the use of satellite and aerial imagery [58]. But even with high resolution data, the question of how to make that data actionable still remains. The delivery of water to vines is almost always done on the block level, where each plant receives roughly the same amount of irrigation. Altering the method of water delivery to somehow be adjustable on the plant level is a problem of enormous scale, which requires replacing the entire irrigation infrastructure and developing a scheme to adjust it when needed. Hardware has been developed to accomplish this in the form of the Portable Emitter Actuation Device (PEAD) [57], which bundles individually adjustable irrigation emitters and a tool to adjust them, however these require solving large logistical problems before they can be put to use. Because plant stress can vary greatly from one vine in the vineyard to the next, and because factors that effect water delivery to plant roots also very from one vine to the next, this problem has yet to be solved and water use efficiency has suffered.

The idea of informing crop nutrient dispersion using crop stress indicators is called precision agriculture, and using water stress sensing to apprise irrigation is called *precision irrigation*. While many well-established agriculture companies, new startup

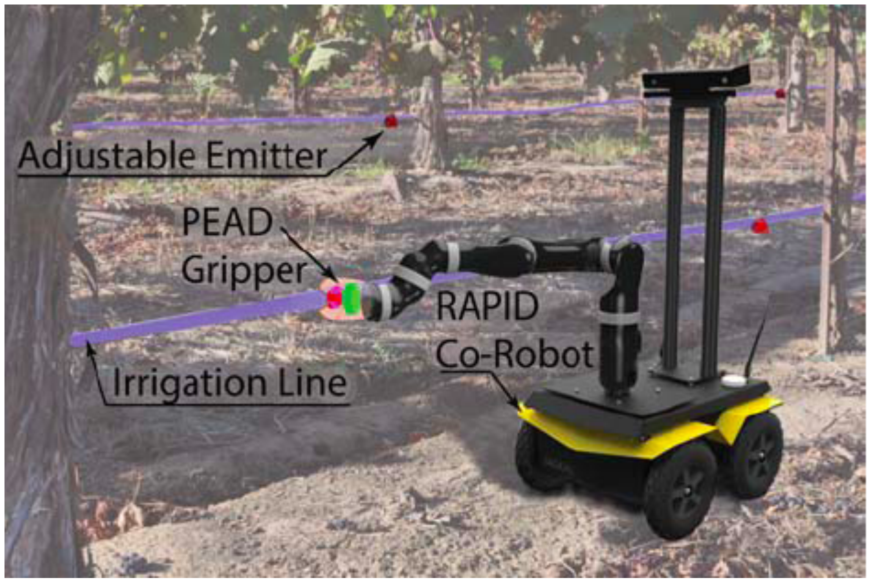


Figure 1.1: Concept art showing how RAPID will operate in the field.

ventures, and researchers are engaged in developing novel methods and technology for sensing data in precision agriculture, there still exists a gap in making that data actionable, especially when applying precision irrigation. However, some groups are working on this. RAPID (Robot Assisted Precision Irrigation Delivery) is a collaborative project from multiple universities with the objective of precisely adjusting the water distribution of irrigation lines on a fine-grain per-plant basis using PEAD on a robotic platform informed by an aerial imagery deep learning network [125]. The hope is that this project will provide a way to close the control loop in vineyard irrigation by providing an adjustable irrigation network and robot capable of making low level adjustments. Using data gathered from remote sensing and direct sensing, RAPID utilizes a processing system designed to decipher the overwhelming amount of data generated into a map of plant water stress, which the co-robots will then interpret into a plan to enact. With the system fully integrated, stress irrigation can be precisely executed without any worry of yield loss or vine damage, thereby saving water and increasing productivity.

There are some challenges to using robots in the vineyard setting, particularly when it comes routing and navigation, and especially when considering the problem of optimizing water usage. Because of the physical structure of vineyards, there are limits to how a robot may travel within it. When applied to a goal, e.g. adjusting emitters in irrigation lines, the nonuniform amount of adjustment required at each point leads to a specialized combinatorial optimization problem where the motion constraints limit the effectiveness of uninformed algorithms. Additionally, the vast size of real-life vineyards means that a planner must be able to parse thousands of data points when calculating an effective route. Finally, because of the organic and outdoor nature of viticulture, circumstances are always changing, meaning prior knowledge may not be correct and predicting future conditions is dubious. These complications make planning paths for robots in vineyards difficult but also interesting to study.

1.2 Overview of Contributions

The purpose of this dissertation is to address the challenges of efficient path planning for robots operating in structured agricultural environments, particularly vineyards. As the goal of routing a ground robot in vineyards is to improve the water use efficiency of the field's irrigation system, the algorithms presented herein are discussed in manners relating to irrigation adjustment, soil moisture measurement, or potential problems associated with field conditions. Regardless, they are also applicable to a wide range of scenarios, for example warehouse robot navigation, and should not be viewed as limited in use to the domain of agriculture. Still, the motivating factors for this dissertation are rooted in improving agricultural efficiency, and therefore experimental assessments of the given methods' veracity use vineyard based data for verification when directly applicable.

There are four main contribution made by this dissertation and the related publications. First, optimal path planning within graphs structured like vineyards is discussed, including multiple different types of deterministic orienteering problems that involve collecting rewards over a graph within a predefined budget. Second, numerous heuristic algorithms are presented, which are able to solve these path planning problems on large-size graphs very quickly, necessary due to the scale of real-life vineyards on which these graphs are based. Third, an extension to stochastic orienteering is discussed, and a set of heuristic algorithms are given which provide time-aware policies for path planning in randomized environments. Fourth, ways produce policies for stochastic orienteering on large-size graphs and speed up computation are presented, such that vineyard environments can be navigated for orienteering even under uncertainty.

The rest of this dissertation is divided into nine chapters which discuss different aspects of efficient robotic routing. Chapter 2 presents an overview of related literature associated with the specific types of problems discussed throughout. Chapter 3 examines single-agent orienteering, or efficient routing for reward collection with a budget, over a new type of graph designed to mimic the structural limitations presented by moving robots within vineyards. Two algorithms are presented which are capable of outputting highly efficient paths that maximize the objective for this problem in very little time, and are shown to be extremely effective at large scale. Chapter 4 extends one of these algorithms to the multi-agent case while also considering how to coordinate all agents in a manner that avoids collisions. The multi-agent methods are shown to be sufficient at routing large number of robots operating in vineyards all at once. Variants of the single-agent routing algorithms which focus on maximizing two reward functions are given in Chapter 5, and shown to be successful like their single-objective counterparts. In Chapter 6, the concept of orienteering is re-tooled to describe problems where travel times between points are random variables with known distributions. A non-adaptive algorithm providing policies that maximize rewards for this new problem is given and evaluated, however it lacks the ability to scale for large problem sizes. Two adaptive improvements to this algorithm are given in Chapter 7 and shown to greatly improve performance on these stochastic problems. A link back to orienteering in vineyards is provided in Chapter 8, whereby a method of reducing the size of paths produced for stochastic orienteering is used in conjunction with the better of the two orienteering algorithms from Chapter 3. This is shown to be highly effective at providing scalability to the stochastic orienteering method. Chapter 9 further improves the scalability of stochastic orienteering by showing how the Lagrangian method can replace the need for a linear programming solver when finding policies and obtain solutions in less time with guaranteed bounds. Finally, Chapter 10 provides some concluding remarks about the methods presented in this dissertation and discusses a few avenues of related future research.

Chapter 2

Related Work

2.1 Robots in Agriculture

The use of robots in agriculture has grown in prevalence over the years. The rise in use of AgTech and precision agriculture has cemented the adoption of automated systems including robots within the industry as necessary components for commercial and environmental success. Typical use cases range from information gathering and remote sensing to automated fruit harvesting and fertilizer application. Additionally, AgTech companies are an emerging niche within the startup world, fueled by recent advances in robotic and computational technology. Led by research teams across the world, agriculture robotics is a growing trend that will continue expanding. For a summary on the opportunities and challenges of robots in agriculture, see [100]. For a more comprehensive review on the subject of agricultural robotics, see [54, 107].

2.1.1 UAVs and Remote Sensing

Some of the earliest and most pragmatic uses of robots in the agriculture industry has been for information gathering and remote sensing. This is due to the fact that remote sensing provides a large benefit in the scalability of gathering information on crop conditions throughout the growing season [136]. Spawned from the many fruitful endeavors of agronomists and ecologists to utilize remote sensing technology, roboticists have incorporated these technologies on their platforms to gather information autonomously. One typical operation involves the collection of image data from aircraft flights above farmland. Usually reserved for human-piloted planes and orbital satellites, imaging over multiple spectra can be done using Unmanned Aerial Vehicles (UAVs) [68].

One example of UAVs used as crop sensors is [133], where the authors used a UAV to collect multi-spectral data and calculate the green area index for wheat and rapeseed crops. The advantage of using UAVs for such data collection compared to conventional methods is that they are less costly, can be flown more frequently, and provide higher resolution image data. However, this last advantage is a double edged sword, as the high resolution data is difficult stitch together and process. This is the problem tackled by [86], where the authors create a single image mosaic map from thousands of individual images of repetitive crop data taken from a low flying UAV. Figure 2.1 shows an example of such a map. Because UAVs fly lower to the ground,

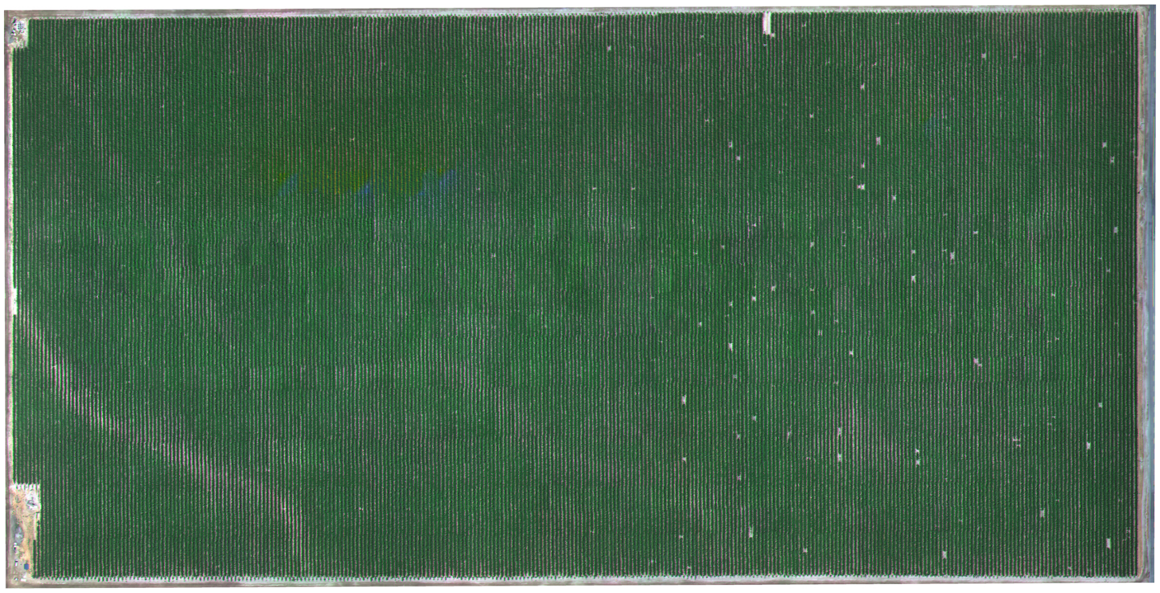


Figure 2.1: An example of an aerial image mosaic of a vineyard. A UAV was flown overhead in a lawnmower pattern to capture hundreds of images which were later stitched together in post-processing. The original of this example is high resolution such that individual grapevines can be distinguished.

they can cover much less ground than manned aircraft or satellites. In order to have a larger footprint with UAVs, research like [13] has utilized teams of UAVs with smart path planning. Others, like [3] use an onboard computer vision system to detect objects of interest such as weeds and to make real-time decisions about where the UAV should fly to next. As the capabilities of UAVs and remote sensing technology continue to improve, these types of robots will become more commonplace out in the field, and help provide farmers with new insights into their crop health and growth.

2.1.2 Applications of UGVs

While flying robots are useful for remote sensing applications, their use cases are limited, and Unmanned Ground Vehicles (UGV) are much more practical for virtually every other operation and are much more capable at addressing the challenges presented by the contemporary agriculture industry [135].

Perhaps the most obvious tasks that would require ground robots for automation are tasks that involve direct interaction with the plants and soil. Robotic fruit harvesting is one such task, where manual fruit picking is used and could benefit greatly from automation, especially for high-value high cost crops. [114] shows the design and integration of a robot capable of harvesting apples without the need for human intervention. In [138], the authors built a unique robot system for harvesting kiwifruit, creating a novel end-effector to safely grab the fruit without harming it, and employed deep neural networks with stereo imaging for fruit detection. However, the kiwi harvester had a success rate of only 51%, highlighting one of the main challenges of robotic harvesting, which is designing a robust and adaptable arm. A study on the viability of telescopic arms versus vacuum grippers for tree-fruit harvesting was presented in [6], where the authors concluded that robots with simple low degree-of-freedom telescoping arms which grab fruit and place them on a conveyor belt are superior to other types of more complex harvesting robots in pear and peach orchards. Overall, fruit harvesting is an important task that is actively being studied.

Besides fruit harvesting, UGVs are also necessary to properly estimate yield for various types of crops. While it is common to assume this task can be performed by UAVs, this is not the case. Indeed, aerial photographs are not very useful for yield estimation since they contain high amounts of occlusion from tree/plant canopies and are too low resolution to reliably detect fruits. Fewer obstructions and higher resolution images are required to distinguish fruits hidden within plant leaves. Instead, ground robots are used to capture images of fruit directly. For instance, [63] used a ground robot with a low-cost camera to estimate the yield of sweet pepper plants. It applied a region-based convolutional neural network to both detect fruit and estimate its ripeness. Neural networks and machine learning algorithms are a common fruit detection mechanism employed by many yield estimation techniques as fruits, such as green apples, are not always easy to distinguish from foliage [12]. Many techniques utilize other tools in addition to or in place of standard cameras. [82] utilizes a RGB-D camera to estimate vineyard yield, which allows for not only grape cluster counting but also for cluster size estimation, thereby allowing a farmer to predict metrics such as average cluster weight and total tonnage yield. For some crops, such as almonds, it is useful to gather data about the orchard canopy volume to give more accurate yield estimates [127], however this requires the use of three-dimensional LiDAR, which greatly increases the cost and computation required. Overall, ground robots are a useful tool to predict crop yield outcomes and will likely become more prevalent as yield estimates increase in accuracy.

There are plenty more tasks that UGVs can perform which are part of the necessary farming operations for successful cultivation. For example, [27] describes a robot that is capable of pruning grape vines, which combines computer vision to build a three-dimension model of each vine and artificial intelligence that chooses which parts of each vine to prune. Another application is in pesticide spraying, which is a recurring task in nearly every growing operation. A remote-controlled robot that can apply pesticide to highly specific targets was developed in [17], which is invaluable due to its potential to drastically reduce pesticide usage with precision application. This is an evolution of a system previously deployed by the same authors in [18], which explored image processing algorithms to detect grapes and foliage with a robotic sprayer, leading to a reduction of pesticide usage up to 30%. Aside from spraying for pests, other robots have been developed that are capable of autonomously weeding orchards and vineyards, such as in [103] where the robot can remove weeds that are growing between vines and within rows without harming the vine trunks. Yet another use of UGVs is shown in [81], which presents a robot that can directly sow

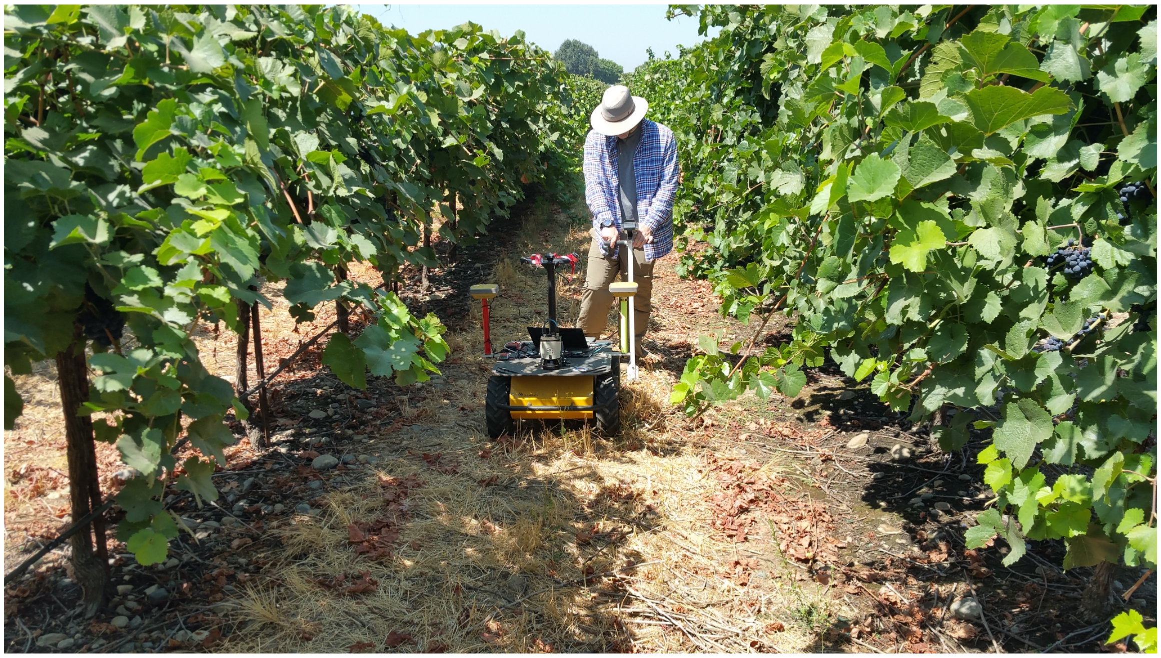


Figure 2.2: A human and a UGV working side by side in a vineyard to measure various indicators of health for the grapevines.

seeds into the ground for farmers. Finally, a vineyard monitoring system that utilizes ground robots is presented in [102] (Figure 2.2 shows a similar robot system). This system makes use of semantic image labeling to efficiently present information about a vineyard's status, such as plant vigor and presence of infestations, to the grower. As evident from the wide array of applications available to UGVs, there is ample room for inclusion of ground-based robots within the agriculture industry.

2.1.3 UGV routing in Vineyards and Orchards

UGVs, having numerous applications within the agriculture domain, naturally require methods for planning while out in the field. There are many reasons for this, such as the need to share the ground with human workers and machinery, the large size of farms and limited battery life or fuel sources of robots. Simply navigating in these environments can be tricky as well. The survey [2] outlines the challenge of localization and mapping in agricultural settings and gives an overview of some techniques specifically developed for use on farmland. For a brief overview of robot path planning and routing algorithms used in agriculture, see the recent review [110].

In many cases, the operation a UGV is performing requires complete coverage of a vineyard or orchard, and so the goal is to optimize variables such as non-working time or distance traveled. One example is [23], where the UGV is a tractor performing one-sided operations (spraying or mowing on only one side of the row being traversed). The tractor must navigate each row between trees twice, with special attention payed to direction and order so that full coverage in minimum distance is realized. A similar strategy is used in [105], where a ground robot takes photos of grape clusters to estimate yield. However, here the goal is to capture close-up under-canopy photos of the grape vines, and thus requires robust localization to prevent mis-identification of vines. They use an Extended Kalman Filter to track vine locations as the UGV navigates through the vineyard. Focusing on planning, a more interesting routing problem is discussed in [143], where the robot must navigate inside of a greenhouse to selectively spray pesticides on certain plants. The authors formulate a multi-objective optimization problem, which is meant to minimize the route time, distance, and rotation while servicing all plants within the robotic sprayer's capacity. What makes this problem interesting is the irregular arrangement of plants in the greenhouse, which takes the form of rows that have openings in the middle. Each of these papers discuss cases where the UGVs perform all their actions in a single traversal of the environment.

In other cases, the UGVs need to make multiple traversals. Typically, some capacity of the robot is limited, such that it requires recharging or refilling multiple times before the operations are complete. Usually the limiting factor is battery life. [109] places recharging stations within a vineyard, so that a robot performing monitoring tasks can recharge without human intervention. The vineyards considered are steep sloped, and therefore energy to travel to each recharging station must be accounted for to prevent complete depletion of battery charge before reaching the station. There can be other limited capacities that require replenishment. For example, the robot in [72] must periodically refill a water tank and replenish its battery when either of these get too low. The battery is depleted from movement through the vineyard, and the water tank is depleted from watering select vines that are determined on the fly to need more irrigation. Since the amount of water needed is not known beforehand, this problem involves accounting for stochastic variables. Another interesting example uses robots carrying fruit bins to and from human harvesters in orchards, minimizing downtime for the pickers [144]. These bins are to be moved and emptied/replaced once they are full, which requires coordination between multiple robots and proper timing to ensure efficient movement.

2.2 The Orienteering Problem

The Orienteering Problem (OP) is a route optimization problem that involves maximizing a reward function for the places visited by the route while keeping to a constraint on the cost of traveling between each place. It is formally defined in section 3.1, and a comprehensive survey of the OP, including its variants and solution methods, can be found in [130, 60].

2.2.1 Route Optimization and TSP

The OP is a special type of route optimization problem. Therefore, it is worth studying other closely related route optimization problems, especially the Traveling

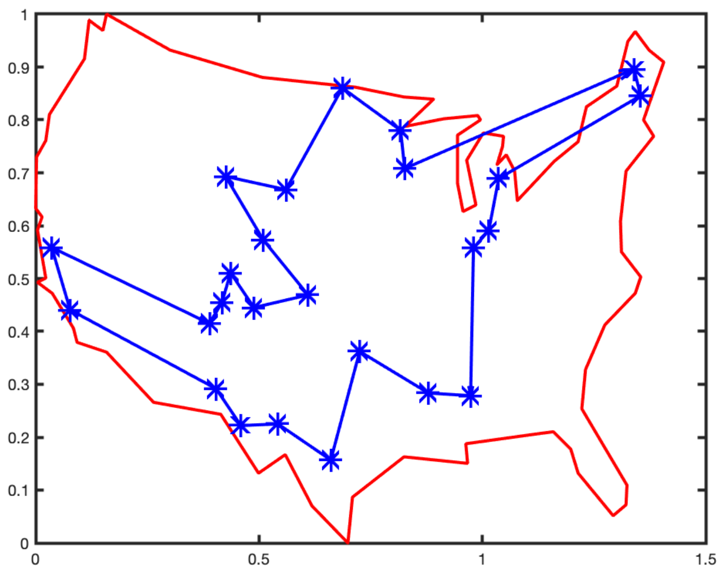


Figure 2.3: An example of a TSP along with its solution. Shown is a map of the United States of America with 25 cities highlighted and the corresponding minimum cost tour that travels to each of them in a circuit.

Salesman Problem (TSP) and the Vehicle Routing Problem (VRP).

Route optimization algorithms can take different forms depending on the criteria or objective of the algorithm. Discretization of the state space and potential actions for each state creates a graph which can be searched for the optimal answer. In routing problems that only account for route length cost, algorithms such as Dijkstra's Algorithm [44] (which is shown to produce optimal results) or A* [64] (which is heuristic based and also optimal when the right heuristic is chosen) are used. In other cases, the goal of the agent is to find a route between multiple goal states that minimizes the total cost of the route. One notable example is the TSP which seeks to find the circuit or tour of minimum length between all vertices in a weighted graph. TSP is known to be NP-hard and therefore computationally expensive to solve for large instances [96]. Optimal solutions for smaller cases are usually found with integer linear programming [41], including Branch-and-Bound or Branch-and-Cut [95] methods, which are extensions to simple tree search algorithms like Depth-first Search or Breadth-first Search. Due to the difficulty of solving NP-hard problems, most choose heuristics methods to obtain approximate solutions. Generally, the more objectives in a problem, the more difficult it becomes to find solutions, and heuristics become favored.

The generalized version of the TSP is the VRP. The VRP seeks to optimize one or more routes from a set of depots to locations defined as vertices on a graph with edges between the vertices. The exact objective varies depending on the particular problem being solved, however, many applications have common goals. These include minimizing the total transportation cost of the vehicles, minimizing the total number of vehicles needed, minimizing variation in costs across all vehicles, maximizing reward for visited locations, and more. For a comprehensive review on different variants of the VRP, see [134]. Like the TSP, the VRP is NP-hard and numerous methods have been developed to solve each variant. Exact methods typically rely on Branch-and-Bound, Branch-and-Cut, and Set-Covering-Based algorithms. See [124] for details on how these work in the context of a few types of VRPs. Aside from optimal solutions, approximation algorithms and heuristics exist for the VRP and its common derivatives. One such approximation algorithm is presented in [10], which obtains a fraction of $1/(3\log^2 n)$ the optimal reward for the VRP with Time-Windows, an important variant of the VPR that restricts when rewards can be collected from vertices. A well known heuristic method is presented in [99], which also solves the VRP with Time-Windows, as well as the Capacitated VRP, the Multi-Depot VRP, and the Open VRP by using the adaptive large neighborhood search framework, providing a number of modifications to a set of routes a keeping those that improve performance. A number of OP heuristic methods are based on similar frameworks such as variable neighborhood search, upon which the adaptive large neighborhood search is based.

2.2.2 OP and solution methods

The OP is another route optimization problem, often described in conjunction with the TSP and VRP. First formally introduced in [126], the OP attempts to maximize a reward function for visited vertices subject to a given budget over traveled edges. Here, each vertex has an associated reward, and costs are defined over each edge between two vertices. Rewards can only be collected once but vertices can be visited more than once if necessary (for example, if the graph is incomplete). Commonly, the OP is rooted, meaning that one or both the starting and ending vertices are fixed, however the unrooted version where no vertices are fixed is also studied. There is a plethora of OP variants, including but not limited to variations with time windows, multiple reward functions, time-dependent travel costs and rewards, etc. as well as solution methods, which are surveyed in detail in [130, 60].

One aspect of the OP that makes it an interesting problem to study is that it is NP-hard. This was proven in [59] by reducing the problem to the Generalized TSP, which contains the TSP as a special case. Many exact solution methods take the form of integer linear programs, which are usually formulated similarly to those solving the TSP or VRP. In fact, it is common to use Miller-Tucker-Zemlin constraints for eliminating subtours [92] on each of these problems, as they are both efficient and easy to implement [43]. In [53], the authors solved the OP using a branch-and-cut algorithm, where they introduce a several families of inequalities and families of cuts that work to find the optimal solution relatively quickly. The downside of using exact solvers for the OP is that they require run time exponential in the size of the

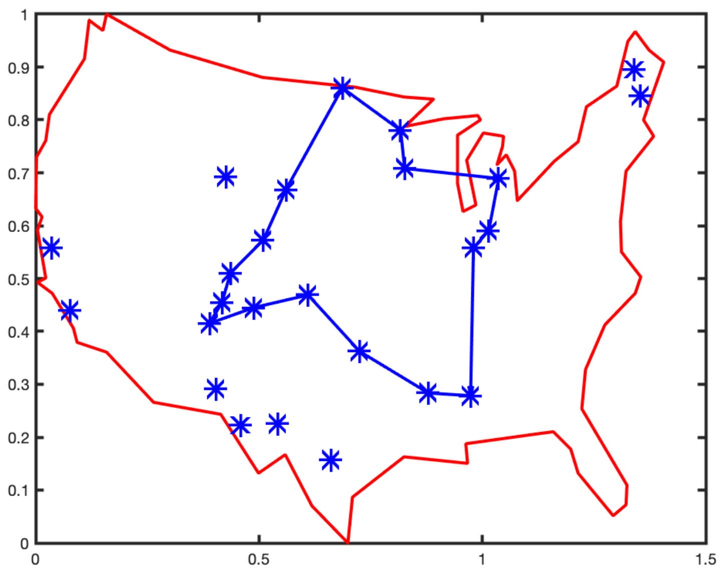


Figure 2.4: An example of a OP on the same graph as in Figure 2.3. Each city has a reward value of 1, with the start and end vertex set to the same city. Here, the budget is limited and thus the tour can only visit a subset of the available 25 cities.

problem. Thus, the crux of these OP solvers is the algorithmic complexity, which limits practical problem sizes to graphs with less than 1000 vertices.

The inherent complexity of the OP means that exact solution methods are not ideal for solving large problems. Thus, approximation algorithms are important in the development of practical solution methods. In [22], the authors show that the OP is also APX-hard, meaning that there is a constant factor $c > 1$ such that approximating the OP to within c is also NP-hard. The authors presented a 4-approximation algorithm for the rooted version, however that was quickly improved to a 3-approximation in [10]. Currently, the best known approximation algorithm for the rooted OP gives a $(2 + \varepsilon)$-approximation solution with a run time of $n^{\mathcal{O}(1/\varepsilon^2)}$, where n is the number of vertices [36]. There exists a $(1 + \varepsilon)$-approximation scheme for the specific case of fully connected planar graphs [37], however this special criteria makes the scope of applicability limited and not useful for the purposes of this work. The unrooted version of the OP was shown to be related to the k-TSP in [7], thus allowing the use numerous approximation schemes for both problems. In [56], a 3-approximation algorithm with polylogarithmic run time was presented. This was later improved to a $(2 + \varepsilon)$-approximation in [36]. Accordingly, approximation algorithms for the OP are useful for obtaining solutions quicker than using exact methods on moderately sized problems.

2.2.3 Heuristics for the OP

While exact and approximate solution methods for the OP exist, using them takes lots of time. This is due to the hardness of the problem causing intractable growth in the solution space, meaning that solving a single large instance or multiple instances takes lots of computational power. Because of this, numerous heuristic algorithms with low time-complexity have been proposed in literature which provide very good solutions at the expense of no guarantees for bounded results.

General purpose heuristics (that is, heuristics that work on a variety of OPs without special requirements) often rely on assumptions made about the metric space to inform the search process. One example is the Center-of-Gravity heuristic presented in [59], which assumes euclidean distances between vertices. This can be useful when graphs are fully connected and lack other types of structure, however it ignores the possibility of obstacles or edges who's lengths are not equal to the distance between the two surrounding vertices. This makes using heuristics with these types of assumptions difficult for real-life problems unless the problems are simplified to fit those assumptions. Other general purpose heuristics instead rely on iterative or Monte Carlo methods to find good solutions, but suffer from having too many parameters that greatly effect results and run time. The Four-Phase Heuristic in [101] is one such algorithm, which takes 5 parameters, modifying things like when phases are activated or when stopping criteria is reached, with varying effectiveness on the end solution and processing time. Typically, good results require more iterations and stricter stopping criteria, leading to long run times. Another well known heuristic which may need a large number of iterations is the S-algorithm [126]. This is due to the stochastic nature of the algorithm, where routes are created by inserting vertices using a weighted random sample according to reward-over-distance measurement. The more random routes are created, the more likely it is that one of them will be optimal or close to it. However, like most other iterative approaches, the number of iterations needed grows as the size of the problem grows. Thus, these heuristics can stretch beyond the capabilities of exact and approximation schemes, but not too much further ($> 10,000$ vertices) due to lack of scalability.

In some domains, such as agriculture, OP graphs may contain tens of thousands of vertices, and therefore general purpose heuristics with long run times are not useful. Here, it is necessary to utilize domain-specific heuristics to obtain meaningful solutions. This fact is highlighted in [121, 115], which show heuristic methods developed to work on aisle graphs (also called Irrigation Graphs in the former) that will provide quick solutions to problems with graphs containing upwards of $100,000$ vertices. In practice, these domain-specific heuristics offer the best balance between reward collection and computation time compared to other algorithms, however they come at the expense of generality (they only work on aisle graphs) and do not provide any sort of guarantees.

2.3 The Team Orienteering Problem

One important variation of the OP is the Team Orienteering Problem (TOP), which has wide applicability to many real-life environments. In the TOP, each agent has a defined budget independent from the other agents and accrues cost from traversing edges separately. Rewards are only collected once from each vertex, regardless of how many agents visit it. The goal remains the same, to maximize the total collective reward. As mentioned earlier, most variations of the OP are NP-hard and APX-hard, including the TOP and its sub-variants. Therefore, this problem is at least as difficult as the OP, and the complexity in number of possible routes grows even faster than the OP because of the team of agents. As with the OP and other route optimization problems, the TOP can be formulated as rooted or unrooted, but the rooted case is more common as it is more practical for a team of agents - be they people, robots, vehicles, etc - to all begin from a single base location. A formal definition of the TOP is supplied in section 4.1.

2.3.1 TOP Solution Methods

Although the first work to mention the TOP by name was [34], the TOP was first studied in [29], where it was called the Multiple Tour Maximum Collection Problem and was shown to be NP-hard. Solving the TOP optimally requires methods similar to those used for the single agent OP, as the OP is evidently a special case of the TOP. Like before, exact solution methods commonly employ integer linear programming, usually relying on the same Miller-Tucker-Zemlin constraints for eliminating subtours [92], though these constraints are needed for each individual agent. One of the first optimal solvers is given in [30], which adopts a set-partitioning formulation that uses column generation and constrain branching to efficiently search for a solution. Here, the version of the problem solved requires all agents to start and end at the same vertex, but allows them to have different budgets. A Branch-and-Price algorithm is given in [28] which also decoupled the depot vertex so the routes were allowed to start and end at different vertices (though they had to be the same for every agent). Both of these formulations were limited in the size of problems they could handle, with the authors testing up to 100 and 102, respectively. A more recent approach was developed in [76], which applied a branch-and-price technique to optimally solve a number of previously non-optimized problems with up to 102 vertices. Around the same time, [47] gave an integer linear program formulation with a number of variables polynomial in the number of vertices which used a cutting planes technique to solve the TOP, and were able to optimally solve a few more previously unsolved benchmark problems with up to 100 vertices. However, some benchmarks for the TOP go up to 500 vertices [60]. Evidently, the TOP is much more difficult than the OP and optimal solution methods are more limited in the size of problems they can solve.

Due to the intrinsic difficulty of the TOP, exact solvers suffer from runtime complexity issues. Unfortunately, the same properties that make the TOP difficult to solve exactly also make it difficult to approximate. After an extensive literature review, only one approximation scheme was found to exist. First proposed in [137] for the TOP and later expanded to the Generalized TOP in [140], the authors consider the case where each visitation of a vertex by agents subject to a submodular function, such that more agents servicing a vertex provides diminishing returns. The algorithm achieves a $(1-(1/e)^{\frac{1}{2+\varepsilon}})$-approximation, where ε is a given constant $0 < \varepsilon \leq 1$. This being the only known approximation scheme, the lack of available methods hints at the intrinsic difficulty of efficiently planning multiple routes at once.

2.3.2 Heuristics for the TOP

Lacking quicker-than-optimal approximation algorithms for the TOP means that much of the focus for obtaining good solutions comes in the domain of heuristics. Plenty of heuristics have been developed for the TOP [130, 60], and most fit into the category of general case heuristics, which are meant to be useful in many circumstances. Some are built upon randomization procedures, such as Simulated Annealing [87] and Ant Colony Optimization [74], which build and modify tours by stochastically choosing the next vertex in a path and where the adjustments are made until improvements are no longer seen. These techniques have the advantage of making immediate improvements in the beginning but decay in advancement speed quickly. Often times, they get stuck in a local optimum and fail to find better solutions. Another variety of heuristic used is the multi-phase meta-heuristic, which involves the conjoining of multiple simple heuristics into a coherent procedure that evolves tours iteratively. Techniques built upon this idea usually rely on the same set of heuristics to construct and update agent paths, such as insert, remove, replace, 2-opt (k-opt), and swap. Depending on the order in which these heuristics are performed and the hyper-parameters used, different results are obtained. Guided Local Search (GLS) in [128] falls into this category, as does Skewed Variable Neighborhood Search in [129], and while they were designed to give good solutions quickly, they lack the scalability required for large problem sizes and are rather inefficient. Overall, the TOP is well studied and there are plenty of heuristics to choose from when it comes to the general case TOP, however many of these are still limited in scalability.

It is worth mentioning that there are not many heuristic algorithms that are built to solve the TOP in specific cases. Most work has been done on general algorithms that will work on any graph, usually with the requirement of euclidean metric space, but few on specialized types of graphs or graphs in different metric spaces. The development of heuristics limited to unique cases has potential advantages, as they can utilize extra information that is encoded implicitly in the graphs that would otherwise be overlooked. This can result in solutions closer to optimum, quicker run times on larger graphs, or both. Thus, it is necessary to explore these options instead of relying on indistinct and potentially ineffectual general heuristics. One example is a heuristic presented in [14], which is a learnheuristic that incorporates machine learning to provide estimations of travel time between vertices on problems containing up to 150 vertices. This is important because travel time is dependent on the order of visitation, as in this case drones need to follow defined laws of motion. There are many instances like this where either motion constraints or problem size make solving them impractical or impossible with more general methods.

2.3.3 Multi-Robot Coordination

When considering the TOP it is often necessary to consider interactions between the robots as well. This is related to the multi-robot motion coordination problem, where constraints limit the availability of vertices or paths between vertices to one robot at a time. A typical approach to handling this type of constraint is to use a space/time composition to resolve conflicts and schedule robots such that they do not interfere along shared routes [97]. In fact, multi-robot path planning with coordination constraints has been proven to be NP-complete even on planar graphs by [141]. Additionally, in [9] it was shown that two-dimensional grid graphs, which are a common form of discretization in motion planning problems, retain NP-hardness. There exists, however, some exact and heuristic methods solving this problem. [142] gave an optimal solution method based on integer linear programming and was able to provide additional heuristics to improve the computational performance of this method while giving up only a small amount of optimality loss. Still, the intractability of this type of problem means that it remains tough to solve, and incorporating multi-robot coordination to the TOP is equally difficult.

2.4 Multi-Objective Orienteering

Often times, the (T)OP is not sufficient to plan for all the requirements of the system. In many cases, a more appropriate representation of the problem is the Multi-Objective Orienteering Problem (MOOP), which needs to account for multiple objective functions, typically as a set of extra rewards or costs defined for each vertex/edge. A subset of the MOOP is the Bi-Objective Orienteering Problem (BOOP), which restricts the number of objective functions to two, or one additional reward function. In this context, an objective function is taken to mean a function which is maximized or minimized. Therefore, the BOOP formalizes a problem where an agent must compute a route that collects two separate rewards and maximize both. The BOOP is described in detail in [112], and is shown to be NP-hard as an extension of the classic OP. A more formal definition for the BOOP is given in section 5.1.

2.4.1 MOOP Solution Methods

Multi-objective combinatorial optimization problems have been studied for a long time. Discussion has centered on creation of a Pareto frontier, which defines the boundary set that obtains maximum values for each objective when making acceptable trade-offs [93]. Such a set is necessary, as there is often no unique solution to a multi-objective problem that fits all criteria. For example, in a two objective problem, maximizing for one objective only may not generate sufficiently large values for the other objective. Instead, a Pareto frontier is used to explore the space between both objectives. Because a large portion of the solution space must be explored to return a Pareto frontier, multi-objective combinatorial optimization problems are in general more difficult to solve than their single objective counterparts. In order to produce a Pareto optimal solution within reasonable time, similar but different techniques are used than in single-objective solvers. For example, [98] uses a generalized branch-and-bound framework that makes comparisons of bound sets instead of numerical values, allowing for the selection of Pareto sets at each iteration, in order to solve bi-objective integer programming problems. The authors also apply this algorithm to the bi-objective TOP with time windows, and show it is effective at solving orienteering-type multi-objective problems quicker than other methods. However, being integer program based, this method is limited to relatively small problem sizes. Some work has been done in the bi-objective optimization domain regarding approximation schemes, though it is sparse. [52] is one of the few found after a thorough literature review of the subject. In it, the authors give two approximation methods which generate ε-approximation efficient sets of the Pareto frontiers by solving lexicographic subproblems at each iteration with ϵ-constraints. These methods are based on modifications to two different types of solvers, one exact algorithm for bi-objective combinatorial optimization problems and one approximation algorithm. They then compare results using a bi-objective version of the TSP with profits.

2.4.2 MOOP Heuristic Methods

There are plenty of heuristic methods developed for the MOOP and BOOP, mostly as modifications to existing OP heuristic solvers. Commonly used methods are meta heuristics of various types, including local and variable neighborhood search algorithms, evolutionary algorithms, and ant/bee colony algorithms. Often times, the motivation for developing these heuristics is for specific applications, such as tourism or work scheduling, and many design choices made reflect this.

Variable neighborhood search methods are one of the more common types of heuristics used. One of the algorithms described in [112] is of this type (solving the BOOP), and integrates a path relinking procedure that combines previously found solutions to create additional efficient solutions and develop a Pareto frontier. In [48], the TOP is solved with soft constraints, where agents are allowed to violate their budget but with an incurred penalty. Here, a biased-randomized search is used, where the search space is directed non-uniformly toward new solutions, turning the deterministic variable neighborhood search into a probabilistic approach. Another method, using the greedy randomized adaptive search procedure with iterative local search, is given in [104] that solves the BOOP with an additional budget constraint. This problem assumes each vertex has a fee associated to it, and the total fees from vertices visited must not violate the extra budget, which makes for an interesting and unique problem. Finally, a BOOP with time windows is solved in [90], giving a set of non-dominated solutions much like a Pareto set which is used as a starting point at each iteration to thoroughly explore the Pareto frontier.

Another type of approach used frequently in VRP and OP heuristics is the evolutionary algorithm. There is no exception for the multi-objective variants. For example, [26] uses a dynamic evolutionary method for the team VRP that generates a proportionally distributed set of routes with relatively equitable lengths and numbers of visited vertices, all while collecting multiple objectives. The same group of authors also contributed [25] solving an online version of the bi-objective VRP, where new vertices are randomly added to the problem as the vehicle traverses its route. The algorithm used here is similar to that used in their multi-objective approach. Regarding orienteering, [15] introduced a hybrid approach for the team MOOP that combines local search strategies with evolutionary path crossover and mutation procedures to generate efficient solutions in a Pareto set. Another approach, this time for the team MOOP with time windows, was given in [66] which combined constraint programming and an evolutionary algorithm with decomposition to generate many objective vectors that are combined to produce better solutions.

Cousins of the evolutionary algorithm, the artificial ant colony and bee colony approaches also have widespread use in MOOP and BOOP heuristics. The other algorithm described in [112] is of the ant colony type, and makes use of the same path relinking procedure within the ant colony to develop the Pareto frontier. [38] uses ant colony optimization to generate solutions for the MOOP with time windows. It does this by decomposing the problem into single-objective sub-problems using scalar weighting, and allowing the ant colony to find efficient paths for each weight to create a set of usable solutions. [91] introduces two approaches to obtain solutions for the time-dependent MOOP, one an ant colony approach using insertion-based local search for generating new paths, and the other a memetic approach (a type of evolutionary heuristic). Lastly, an artificial bee colony algorithm was given in [89], which was designed for the BOOP. It uses agents or "bees" with different jobs to conduct a form of swarm intelligence and find non-dominated Pareto solutions to the problem. Overall, these insect inspired algorithms provide unique and capable solutions to multi-objective vehicle routing problems such as the MOOP and BOOP.

2.5 Constrained Markov Decision Processes

One common problem with standard route optimization problems is that they only consider the case where outcomes are deterministic. For example, Dijkstra's Algorithm [44] can find the shortest path between vertices in a graph, however each edge is assumed to have a deterministic length. If that assumption is removed, then the optimal solution is no longer a path but a policy that dictates what route to take based on previously incurred costs. Accordingly, real world environments contain elements of randomness that are impossible to account for in route planners that only find solutions to deterministic problems. When accounting for stochasticity in route planning, the Markov Decision Process (MDP) is a useful tool. The MDP provides a model and decision making framework for handling problems where an agent must account for randomness in outcomes when presented with a set of choices.

2.5.1 MDPs and Extensions

One of the first discussions on MDPs as they are currently know was in [16]. The MDP derives its name from and is an extension of Markov Chains. In addition to states and uncertain transitions between states, MDPs also contain rewards for arriving at states and actions an agent can take to influence transitions to future states. The goal of an MDP is to find a policy, or action for each state, that influences state transitions such that the overall reward is maximized. [16] introduced what is now called Value Iteration as a solution method that converges to the optimal policy. This is a dynamic programming method that iteratively updates the value function of taking each action in every state until the value no longer changes (or reaches a suitable stopping criteria). Then, a policy is derived from the value function. [65] proposed a slightly different but still optimal method called Policy Iteration, which instead derives a policy at each iteration and is updated until there are no more changes to the policy. Both of these are dynamic programming solutions to MDPs which must at each iteration sweep over the entire state space and perform calculations for every action in each state. Because of this, MDPs are said to suffer from the "curse of dimensionality" which tends to make large problems computationally intractable due to the volume of states and actions that need to be explored. This difficulty exists in linear programming methods for solving MDPs as well. In fact, evidence points to dynamic programming methods being more computationally efficient than linear programming [79]. Therefore, Value Iteration and Policy Iteration are still the dominate choices when it comes to finding optimal policies.

MDPs are well suited to decision making problems with discrete stochastic control. However some common extensions allow the framework to be more applicable to a broader class of problems. One of the most useful extentions is the Constrained MDP (CMDP). CMDPs contain everything MDPs have, but additionally have objective (cost) functions that give extra costs for visiting a state or taking an action. A given policy is required to meet defined bounds on the extra objective functions to be valid. Thus, the optimal policy for a typical CMDP is one that maximizes the reward function while obeying the bounds on each additional cost function. In depth information on CMDPs can be found in the definitive source [4]. The CMDP has been researched nearly as long as the ordinary MDP, and one of the first to present a solution method was [42], which provided a linear programming approach that uses occupation measures of each state/action pair as decision variables. This approach has been extended numerous times, and its most commonly used forms are the expected average cost for finite and infinite CMDPs [4]. In these forms, the computed policy maximizes the reward function and keeps the cost functions bounded to the given constraints in expectation. Another well known method to computing a CMDP policy is the Lagrangian approach. This was introduced by [21] and shown to produce an optimal policy for the average cost case finite state space form in [113]. The disadvantage to the Lagrangian approach is that that it does not intuitively extend beyond a single constraint, however it has been studied (see [5] for an example). There are interesting variations on CMDPs that use different types of constraint criteria. For example, [73] develops a Variance-Constrained MDP that aims to bound the variance of the reward function for the resulting policy. In [24], the Risk-Constrained MDP was developed to bound the Conditional Value-at-Risk of the optimized function using tail values of probability.

Regardless of the type of constraint criteria used, finding policies on CMDPs with large problem sizes is still a computationally expensive task, with many practical problems being intractable. Therefore, heuristic and learning methods are very popular. [84] applied the technique of Monte Carlo Tree Search to solve very large CMDPs based on a multi-objective variant of the Atari game Pong. Monte Carlo Tree Search is advantageous because it allows for rapid exploration of the state space while also exploiting previously found good policies. The authors take it a step further and combine this with the UCT algorithm [78] to create a cost constrained version that converges to the optimal stochastic action. Another use of the Monte Carlo Tree Search technique is presented in [8], where the authors use it to find policies for the Chance-Constrained MDP, and showed its capability to find policies for extremely large state spaces (10^{13}) in only a few minutes. Despite the success of randomized searches for CMDPs, more principled approaches to solving large problem instances are still desirable. One in particular is given in [51], which clusters the state space into "macro states" to create a hierarchical CMDP. Each macro state has its own policy generated under specific conditions, and the policies are merged together to create a single policy for the entire state space. This has the advantage of splitting large CMDPs into smaller problems which are easier to solve and take less computation time.

2.5.2 Problems Solved using CMDPs

CMDPs are useful for tackling a myriad of different problems, despite being computationally limited in the size of state space that can be considered. Because policies are designed to allow agents some level of autonomous control, CMDPs have found widespread adoption in robotics with safety or performance constraints. For example, [33] presents the the Rapid Multirobot Deployment problem, in which a set of robots is tasks with reaching multiple goals within a pre-assigned time limit. Safety of the robots is also considered, and thus a CMDP is used to minimize robot risk while constraining the amount of time needed to perform the task. This problem was later extended to a stochastic version in [40] with a robotic swarm whose objective is to position at least one robot at each target site, under the assumption that some robots may fail. [106] discusses a similar problem where a team of robots needs to attain a set of goals while obeying multiple types of constraints. This is formulated as a type of VRP which is solved using a CMDP policy to control which robots perform certain tasks and how to coordinate tasks so that they do not violate constraints. Another use of CMDPs for robotic teams is shown in [77], where multiple solar powered UAVs are used as access points in a wireless network. A CMDP is evaluated in the context of network dynamics using Deep Reinforcement Learning to create a policy that maximizes long term network capacity while preserving energy sustainability of the UAVs. Along similar lines but without autonomous vehicles, [139] uses a CMDP to formulate a device to device communication mode selection problem. A cellular user requires a minimum data rate while the network must enforce a peer to peer deadline constraint. The authors find policy that optimally controls network modes using the Lagrangian technique for CMDPs. The problem of risk mitigation for driverless cars has also been studied, using Chance-Constrained Partial Observable MDPs in [67]. Here the authors use a specialized forward search heuristic called RAO* [108] combined with continuous maneuver modeling using probabilistic flow tubes to remove the limitations of discrete state and action spaces when creating policies. The RAO* algorithm creates a search tree in belief space prunes risky behaviors to keep the search reasonable. This is in contrast to how a Monte Carlo Search Tree on the same problem type [85] typically behaves. Finally, Multi-Objective planners have also been developed which utilize CMDPs to model very complex problems, as evidenced by [50] which uses Linear Temporal Logic to semantically determine and complete multiple subgoals within satisfaction probabilities. Overall, constrained formed of MDPs have proven extremely useful for multiple types of problems and they clearly have an important place in any planning toolbox.

2.6 The Stochastic Orienteering Problem

The OP, as it is usually studied, is a deterministic problem. However, many real world situations that are useful to model as OPs are, in fact, stochastic in nature. For instance, one might consider a tourist visiting different attractions of a city in a day. The time they have is limited and they want to visit some attractions more than others, so they need to optimize their route to get the most enjoyment out of their day. However, travel time between attractions is stochastic due to traffic, time to experience each attraction is stochastic due to lines and sojourn duration, and the reward for visiting attractions is stochastic because the tourist may enjoy it more or less than expected. To effectively model this type of scenario requires the scrutiny of the problem with randomness considered. Thus the Stochastic OP (SOP) is conceived. A formal definition for the SOP, as well as its chance constrained sibling, is given in section 6.1.

2.6.1 Solution approaches for the SOP

While the OP has been studied for a long time and significant research has been devoted to it, the SOP is much less well studied. First introduced in [31], the authors examined the case with uncertainty in both the cost to traverse an edge (stochastic travel time) and the cost to visit a vertex (stochastic service time). The solutions they presented were either heuristics for the general case or exact algorithms that are only applicable to certain types of graph topologies. Unfortunately, the SOP is very difficult to solve and designing algorithms to solve the general case problem exactly has proven elusive. Like its parent problem, the OP, the SOP is NP-hard and APX-hard. The first approximation methods designed for some versions of the problem were given in [61]. For the case of stochastic travel and service times, the authors give two types of approximation algorithms, one for adaptive (policy driven) solutions and one for non-adaptive solutions. They show the existence of an adaptivity gap, with the non-adaptive solution obtaining at least $\Omega(1/\log\log B)$ of the optimal adaptive reward, where B is the budget of the given problem, with the adaptive approach obtaining a $\mathcal{O}(\log\log B)$-approximation. Also shown is a non-adaptive $\mathcal{O}(\log n \log B)$-approximation, with n as the number of vertices, for the version of the problem where vertex rewards are stochastic as well. In [62] the same authors extended their approaches to directed graphs but were not able to improve on the metrics. However, the related literature [11] was able to improve on the adaptivity gap, proving a lower bound of $\Omega(\sqrt{\log\log B})$. This bound improves on the approximation bound provided earlier, but the authors were not able to present an algorithm that achieves it.

Like with the OP, heuristic methods are useful when quick solutions are required or when problem sizes make the use of approximation algorithms intractable. As mentioned earlier, [31] gave a heuristic for the SOP that is a modification to the popular Variable Neighborhood Search algorithm for the deterministic OP. The modification takes in a function that determines the overall cost of a computed route by summing the arrival time distributions for each vertex to obtain a single distribution and then uses the mean of that as the total cost. [49] presents two solutions to the SOP, one method solving a sample average approximation version with mixed integer programming, and one method using heuristics. The mixed integer programming method only works for small instances, and requires optimizing for a tour and the amount of reward not collected, which they call recourse, by the tour. The heuristic technique constructs stochastically a number of solutions using a score measure based on reward recourse and estimated edge costs, then improves on the solutions with a few local search techniques and chooses the best route as a solution. These heuristics are not adaptive, however, so routes produced do not change to maximize rewards as the path realizes costs different from those that were estimated. [45] presents three heuristics, one that is not adaptive and two that are. The adaptive ones differ in that one updates the edge costs to realized costs online, while the other determines a dynamic policy beforehand. The advantage of these methods is shown in the difference of reward for each level of adaptivity.

2.6.2 The SOP with Chance Constraints

One aspect of stochastic problems that is often over-looked when solving the SOP is the chance of failure. The solution methods presented earlier all seek to maximize the expected reward gathered for the computed path or policy, however none consider the risk associated with the given solution. These solutions usually have an expected failure rate of approximately 50%, and therefore should be considered risky, especially for real world problems where failure can be catastrophic. This is why the SOP with Chance Constraints (SOPCC) was created, to solve real world problems while also mitigating risk.

One of the first works to study the SOPCC was [83], which looked at a chance constrained version of the Dynamic SOP, a generalization of the SOPCC where edge costs vary across time. The authors formulate a few ways to approximate the completion probability of an individual path and use these techniques to develop a local search heuristic algorithm that incorporates the completion probability as part of the utility metric. In [131] a mixed integer linear program is given which solves the problem using a sample average approximation and linearizing the constraint. This gives a decent deterministic approximation for the solution to a SOPCC however the results do not have an approximation bound. The sum of work from [83] and [131] resulted in [132], which was able to extend the mixed integer linear program with sample average approximation to the Dynamic SOPCC and compare the performance with the local search heuristic on problems up to 63 vertices and 40 samples in size. Another variation on the SOPCC was studied in [71], called the Team Surviving Orienteers Problem, which considered edges to have survival probabilities instead of costs. A team of agents is deployed to collect maximum reward while ensuring that at least one agent survives within a probability constraint. The authors gave a greedy algorithm with an approximation guarantee within a factor $1 - e^{-p_s/\lambda}$ of the optimum, where p_s is the per-robot probability of survival, and $1/\lambda \leq 1$ is an approximation factor of an oracle routine for the OP. Unfortunately, the SOPCC is a rather sparsely studied problem and a thorough literature review did not reveal many more interesting variants or solution methods.

Chapter 3

Single Robot Orienteering in Vineyards

In this chapter, the concept of orienteering in vineyards is discussed and heuristic algorithms are developed to solve these types of problems. The work shown here was originally presented in [121].

3.1 OP Background

In this section, the OP is formulated and two solution methods for the general case problem are given, one based on integer linear programming and one randomized heuristic method, which are later used as benchmarks for performance comparison.

3.1.1 Problem Definition

For simplicity and completeness, we first define a path. Let $G = (V, E)$ be an undirected graph containing a set of vertices $v \in V$ and a set of edges $e \in E$ connecting the vertices. The edge $e_{i,j} = E(v_i, v_j)$ connects vertex v_i to v_j, and there is only one such edge that does so (if more than one edge exists for this junction, the edge with higher cost can safely be discarded). A path $\mathcal{P}$ in G is an ordered set of vertices that are connected sequentially by valid edges. The path starts at some vertex and visits a number of vertices until reaching the last in the set, $\mathcal{P}(i) \in V$ where $1 \leq i \leq |\mathcal{P}|$. This also implies a specific set of edges which the path traverses, where the edge connecting $\mathcal{P}(i)$ with the next vertex in the sequence is denoted as $\mathcal{P}_e(i) = E(\mathcal{P}(i), \mathcal{P}(i+1))$.

The OP, discussed in section 2.2, will now be described in detail. A reward function $r : V \to \mathbb{R}_{\geq 0}$ associates each vertex $v_i \in V$ in G with a reward $r_i = R(v_i)$, and a cost function $c : E \to \mathbb{R}_{\geq 0}$ associates each edge $e_{i,j} \in E$ in G with a deterministic cost $c_{i,j} = C(e_{i,j}) = C(v_i, v_j)$. Given are a start vertex $v_s \in V$, a goal vertex $v_g \in V$, and a budget $B \in \mathbb{R}_{\geq 0}$. The OP asks to find a path $\mathcal{P}$ that starts at v_s and ends at v_g, which maximizes the sum of collected rewards and has a cumulative cost no more than B. The reward of the path $R(\mathcal{P})$ is the sum of rewards for all uniquely visited vertices. That is, each vertex visited by the path can contribute its reward only once. This is important in problems where G is not fully connected, as it is possible to build paths on G that necessarily visit some vertices more than once. The cost of the path $C(\mathcal{P})$ is the sum of costs for all traversed edges, where an edge traversed more than once is counted each time. Again, this is important in cases where G is not fully connected.

The version of the problem described here specifies both the starting vertex v_s and the goal vertex v_g, and thus is the rooted OP. In many circumstances, v_s and v_g are coincident such that the resulting path $\mathcal{P}$ begins and ends at the same place. This is usually called a "tour" in literature. In the following, the term "tour" refers to any path in G where $v_s = v_g$ and the term "path" refers to any path in G including tours. Regardless of the type of path specified, the OP remains NP-hard [59]. Moreover, without loss of generality, it can be assumed that the graph G is complete. If this is not the case, additional edges can be supplied with cost equal to shortest path between the vertices they connect.

3.1.2 Linear Programming Formulation

As mentioned earlier the OP can be formulated as an integer linear program which can then be solved to produce an optimal solution. Here, one version of this formulation derived from [130] is shown and will later be used as a benchmark comparison method. The notation from above is used and extended. There are two types of decision variables used, $x_{i,j}$ describing the inclusion of edge $e_{i,j}$ in the path (1 if it is included in $\mathcal{P}$ and 0 otherwise), and u_i describing the position of vertex v_i in $\mathcal{P}$ such that $\mathcal{P}(u_i - 1) = v_i$.

$$\max \sum_{v_i \in V} \sum_{v_j \in V} r_j x_{i,j} \tag{3.1}$$

$$s.t. \sum_{v_i \in V} \sum_{v_j \in V} c_{i,j} x_{i,j} \leq B \tag{3.2}$$

$$\sum_{v_i \in V} x_{i,k} = \sum_{v_j \in V} x_{k,j} \leq 1 \qquad \forall v_k \in V \tag{3.3}$$

$$2 \leq u_i \leq |V| \qquad \forall v_i \in V \backslash \{v_s\} \tag{3.4}$$

$$u_i - u_j + 1 \leq (|V| - 1)(1 - x_{i,j}) \qquad \forall v_i, v_j \in V \backslash \{v_s\} \tag{3.5}$$

$$\sum_{v_j \in V \backslash \{v_s\}} x_{v_s,j} = \sum_{v_i \in V \backslash \{v_g\}} x_{i,v_g} = 1 \tag{3.6}$$

The linear program can be understood as follows: Equation 3.1 is the objective function, stating that the total reward for each vertex in the path is maximized. Equation 3.2 is the budget constraint, enforcing the sum of costs for all edges in the path to be less than B. The constraint in Equation 3.3 requires that each edge be in the path no more than once (note that when G is a complete graph, an optimal solution path will not need to traverse any edge multiple times). The constraints Equation 3.4 and Equation 3.5 are the Miller-Tucker-Zemlin subtour elimination constraints [92] ensuring a single continuous path. Lastly, Equation 3.6 requires the path start at v_s

and end at v_g. In problems where v_s and v_g are not specified (a.k.a. unrooted orienteering), this constraint can be safely disregarded. After the return of the integer linear program solver, the path is extracted from the inclusion of edges given by $x_{i,j}$ with the sequence of vertices specified by u_i.

3.1.3 S-Algorithm Heuristic

One of the most effective and commonly used general purpose heuristics solving the OP is the S-Algorithm [126]. The S-Algorithm is a stochastic method that uses a Monte Carlo approach to finding a good solution for the OP. A number of paths respecting the budget are generated very quickly, and of them the one with the highest reward is returned as the solution. The algorithm builds each path iteratively, starting from v_s and adding new vertices until there is no more residual budget beyond the amount needed to go to v_g. Assuming the last vertex added to the path is $\mathcal{P}(i)$, the next vertex is chosen randomly from $V \backslash \mathcal{P}$ using a weighted distribution according to a "desirability" metric. Desirability is defined as

$$A(v_j) = \left\{ \frac{r_j + \alpha[B - C(\mathcal{P}) - C(\mathcal{P}(i), v_j) - C(\mathcal{P}(i), v_g)] \cdot n(v_j)}{C(\mathcal{P}(i), v_j)} \right\}^p \tag{3.7}$$

where $C(\mathcal{P}(i), v)$ is the cost of traveling from the i-th vertex in $\mathcal{P}$ to v, α is a weight factor parameter, p is a power factor parameter, and $n(v_j)$ is a "nearness" measure defined by $n(v_j) = \sum_{v \in V} \frac{r(v)}{C(v, v_j)}$. In essence, $A(j)$ returns large values for vertices with high reward that are near to the current vertex and would leave more residual budget if included in the path. The probability that v_j is the next vertex included in the path is calculated as

$$\Pr(v_j | \mathcal{P}) = \frac{A(v_j)}{\sum_{v \in V \backslash \mathcal{P}} A(v)} \tag{3.8}$$

Vertices that have higher desirability are therefore more likely to be included as the next vertex in $\mathcal{P}$. Once the residual budget is too small to include any additional vertices besides the goal vertex, v_g is included in $\mathcal{P}$ and the path terminates.

The reward performance of the S-Algorithm varies depending on the values for each parameter. The most critical of these parameters is the number of different paths generated, as a greater number means a greater likelihood of finding the optimal path. The algorithmic performance of the S-Algorithm is also dependent on the number of different paths generated, as well as the number of vertices in G, and the number of vertices added to each path. The number of vertices in each path is a random number, and it is difficult to determine an expected value for this number using formal analysis. The computational complexity can be written as $\mathcal{O}(n \cdot |V|^2 l)$, where n is the number of Monte Carlo trials (paths) and l is the number of iterations until the budget B is fully spent for each trial.

3.2 Orienteering on Aisle Graphs

While the general case of the OP is known to be difficult to solve, little work has been done determining the hardness of the OP in nonstandard graphs. It may be possible, through clever organization constructing the nodes and edges of a graph, that the OP on a particular graph is trivial or easy to compute. Therefore, it is necessary to prove that any constructed graph on which the OP will be applied does not make the problem impalpable. This section considers graphs resembling the structural layout of vineyards called Aisle Graphs (AGs, also called Irrigation Graphs in [121]).

3.2.1 Bipartite Planar Graphs of Degree 3

First, Bipartite Planar Graphs of Degree 3 must be defined before AGs can be discussed. A graph $G = (V, E)$ is said to be $\mathcal{BP}3$ if it is bipartite, planar, and has degree of at most 3. Bipartite means that the graph can be divided into two disjoint sets of vertices $A \subset V$ and $B \subset V$ such that all edges connect vertices in A with vertices in B. Planar means that the graph can be drawn on a 2-dimensional surface in such a way that none of the edges intersect, and are only allowed to for junctions at vertices. A graph will have a degree of at most 3 when there does not exist any vertex in the graph with more than 3 edges connected to it. On $\mathcal{BP}3$ graphs, the Hamilton Circuit problem, which asks to build a tour that goes through each vertex exactly once, has been proven to be NP-complete [70]. The Hamilton Circuit problem can be reduced to the decision TSP, which asks if there exists a tour of cost no more than a given budget B, by making all edge costs 1 and setting $B = |V|$. Only if there exists a Hamilton Circuit can the answer to the decision TSP be yes, and therefore the TSP on $\mathcal{BP}3$ must also be NP-complete.

The OP requires cost and reward functions over the graph to be defined, and the definition given earlier is used. The Constant Cost OP is a similar to the ordinary OP except all edges share the same cost (often a unit value of 1). The Constant Cost $\mathcal{BP}3$ OP (CCBP3OP) is similarly defined, where $C(e) = k$ for every edge $e \in E$ and k is a constant. Then, the objective of the CCBP3OP is find a path of cost at most B from v_s to v_g that maximizes the sum collected rewards. Theorem Theorem 3.2.1, taken from [121], establishes that this problem is NP-hard. Because the constant cost version is NP-hard, so is the unconstrained cost variant, OP on $\mathcal{BP}3$ Problem (BP3OP).

Theorem 3.2.1. *The CCB3OP is NP-hard.*

Proof. Let $G = (V, E)$, C, B be an instance of the decision TSP on a $\mathcal{BP}3$ graph, and make all costs natural numbers such that for all $e \in E$, $C(e) \to \mathbb{N}$. Then, a graph $G' = (V', E') = G$ with a reward function $R(v) = 1 \forall v \in V$, edges $e = (v_i, v_j) \in E$ with a cost function $C(e) > 0$ can be augmented with $p - 1$ vertices having rewards of $R(v) = 0$ inserted into V' and p edges between v_i and v_j with costs $C(e) = 1$, and a total cost B. This graph is planar with a maximum degree of 3, and can

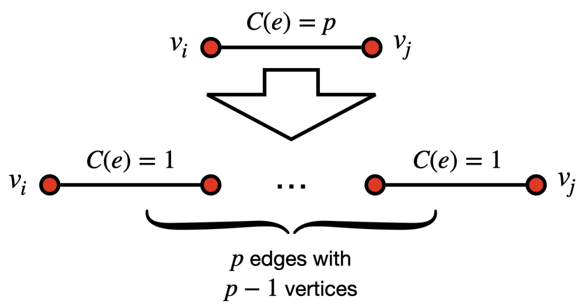


Figure 3.1: The construction of a $\mathcal{BP}3$ from a planar graph of degree 3.

be easily changed to bipartite by introducing a new vertex with 0 reward and two new edges around it with 0 cost in any area of the graph that violates criteria for a bipartite graph, therefore fulfilling all $\mathcal{BP}3$ requirements (see Figure 3.1 for this transformation). The solution to the CCBP3OP will be equal to B only if the answer to the decision TSP problem on the same graph is yes, and therefore the problem is NP-hard as well. □

3.2.2 Aisle Graphs

Expanding on $\mathcal{BP}3$ graphs, AGs, which are applicable to vineyards, can be described. These are designed to capture the motion constraints of vineyards and other arrangements of aisles such as warehouses. In particular, vineyards have predefined areas which any vehicle traversing them must use as roads. In a graph representation, these are the edges. There are two types, the inner edges which cross through the vineyard and are arranged in an aisle-like row structure (hence the name), and the outer edges which connect adjacent rows on both ends. Along each row are stopping points which become the vertices of the graph, connected to other stopping points by an edge on both sides. If the point is located on the outside of the row, the vertex serves as a junction between two outer edges and a single inner edge. Thus this type of arrangement builds a $\mathcal{BP}3$ graph.

Now an AG can be defined using the following set of rules. They are $\mathcal{BP}3$ graphs written as $AG(w, l) = (V, E)$, where w is the number of rows or width of the aisle structure and l is the number of points/vertices in each row or its length. Each vertex is given as $v_{i,j} \in V$ where $1 \leq i \leq w$ and $1 \leq j \leq l$, and the set of edges E follows a number of rules:

- Each vertex $v_{i,j}$ where $1 < j < l$ has exactly 2 edges, joined to $v_{i,j-1}$ and $v_{i,j+1}$
- Each vertex $v_{i,1}$ where $1 < i < w$ has exactly 3 edges, joined to $v_{i-1,1}$, $v_{i+1,1}$, and $v_{i,2}$
- Each vertex $v_{i,n}$ where $1 < i < w$ has exactly 3 edges, joined to $v_{i-1,n}$, $v_{i+1,n}$, and $v_{i,n-1}$
- One edge connects $v_{1,1}$ and $v_{2,1}$

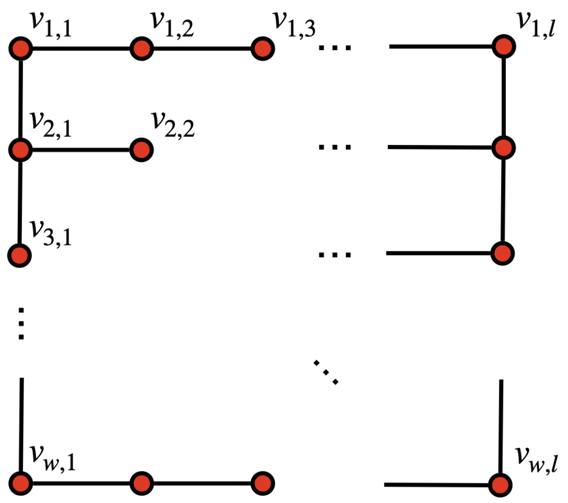


Figure 3.2: The Grid-like structure of the AG. Note the only edges that connect rows to each other exist on the outskirts of the graph.

- One edge connects $v_{w,1}$ and $v_{w-1,1}$
- One edge connects $v_{1,l}$ and $v_{2,l}$
- One edge connects $v_{w,l}$ and $v_{w-1,l}$

An example of the structure of an AG is shown in Figure 3.2. As a graph, it is conceivable to extend the description to include costs and rewards. The cost function C gives positive, real cost to each edge such that $c : E \to \mathbb{R}_{\geq 0}$ and the reward function R gives positive, real cost to each vertex such that $r : V \to \mathbb{R}_{\geq 0}$. With this extension, it is possible to define two special versions of the OP on AGs:

> **Constant Cost Aisle Graph Orienteering Problem (CCAGOP):** Given a graph $G(V, E) = AG(w, l)$ with constant cost function $C(e) = k$; $\forall e \in E$ and reward function R, vertices within the graph $v_s, v_g \in V$, and a constant B, find a path $\mathcal{P} \subset V$ with cost no more than B that begins at v_s and ends at v_g which maximizes the cumulative reward of visited vertices.
>
> **Aisle Graph Orienteering Problem (AGOP):** Given a graph $G(V, E) = AG(w, l)$ with cost function C and reward function R, vertices within the graph $v_s, v_g \in V$, and a constant B, find a path $\mathcal{P} \subset V$ with cost no more than B that begins at v_s and ends at v_g which maximizes the cumulative reward of visited vertices.

These versions of the OP are both NP-hard, as they are special cases of the CCBP3OP and the BP3OP, respectively. Note that in the former [121], these were named the Irrigation Graph Constant Cost OP and the Irrigation Graph Orienteering Problem, however the names are changed here to reflect their applicability to more diverse situations (see [115]).

3.3 AGOP Heuristic Algorithms

Heuristics are meant to accelerate computation of an answer to a problem without sacrificing too much optimality. However, most general case heuristics for the OP were not designed to handle graphs with many thousands of vertices. Since an AG based on a real-life vineyard may be such a graph, heuristics that can handle this size are necessary. For the following algorithms, the start and goal vertices are the same and the path produced is a tour. Both of the algorithms are taken directly from [121].

3.3.1 Greedy Row

The Greedy Row (GR) Heuristic was designed to take advantage of the fact that AGs are neatly divided into rows. The algorithm greedily selects a subset of rows to be traversed as part of the tour using a computed heuristic based on the potential reward to collect and the amount of budget collecting it would use. As a tour is computed, the algorithm tracks the remaining budget to ensure enough remains for travel to the goal vertex from the current position. The pseudo-code for the GR heuristic algorithm is given in Algorithm 3.1.

Algorithm 3.1 Greedy Row Heuristic

```
Input: $AG(w,l)$, $R$, $C$, $B$, $v_s$, $v_g$
Output: $\mathcal{P}$
 1: $\mathcal{P} = \{v_s\}$
 2: for $i \leftarrow 1$ to $w$: $R_i \leftarrow \sum_{j=1}^{l} R(i,j)$
 3: for $i \leftarrow 1$ to $w$: $feasible_i \leftarrow$ True
 4: while $any(feasible) =$ True do
 5:     for $i \leftarrow 1$ to $w$ do
 6:         if $feasible(i,\mathcal{P}) \neq$ True then
 7:             $feasible_i \leftarrow$ False
 8:     for all $feasible_i$ do
 9:         $R'_i \leftarrow R_i / C(\mathcal{P}(end), i)$
10:     $best \leftarrow \arg\max R'_i$
11:     to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $best$ row
12:     append all vertices in $best$ row to $\mathcal{P}$
13:     $feasible_{best} \leftarrow$ False
14: to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $v_g$
15: return $\mathcal{P}$
```

GR works as follows. First, the tour $\mathcal{P}$ is initialized (line 1). Next, the total reward for each row r_i is computed by summing the rewards for all vertices in that row (line 2). Then every row is marked as feasible, which means that the row has not yet been visited and that there is enough budget to travel from the current vertex $\mathcal{P}(end)$, to and through the row, and to the goal vertex (line 3). A loop is entered where the tour is built (line 4). Each row is checked for feasibility and marked if not feasible (lines 5-7), and then the heuristic is computed for each row (lines 8-9). The heuristic is the cumulative reward collected for traversing a row divided by the cost to travel to (up or down the outer edges from $\mathcal{P}(end)$) and across (through the aisle) the row. Then, the row with the highest heuristic value is chosen (line 10) and added to the tour (lines 11-12). The feasibility for this row is then negated so that it will not be considered in future iterations (line 13). This process continues until all rows are marked as not feasible, and the path from the current vertex to the ending vertex is added to the path (line 14).

Overall, this algorithm produces a tour that is guaranteed terminate at the goal vertex because of the feasibility checks. Therefore, the tour produced is always valid, beginning and ending at $v_s = v_g$ and always has an overall cost of less than B. It will terminate after at most $B/l = k$ iterations of the main loop, due to the fact that the tour cost will increase at least l units until there are no more feasible rows, and $k \leq w$. Each iteration of the loop has a complexity of $\mathcal{O}(w)$ since every row is examined. giving the algorithm an overall complexity of $\mathcal{O}(wl + w^2)$. On a square block where $w = l$, the complexity becomes $\mathcal{O}(w^2)$, meaning the algorithm is quadratic on the number of rows in the graph or linear on the number of vertices $|V|$.

3.3.2 Greedy Partial-Row

Using rows as a basis for determining a robot path through an AG is a great starting point for an algorithm because it takes advantage of the graph structure to build efficient tours. The approach, however, lacks awareness of the fact that vertex rewards can be highly localized. It is completely possible that some vertices in a row have high reward values while others in the same row have low reward values. These may be segregated or mixed depending on the characteristics of the problem being solved. For instance, in vineyards each individual plant has unique requirements and soil types can vary spatially. The Greedy Partial-Row (GPR) heuristic was designed to account for this observation by allowing the construction of tours that have the ability to partially traverse rows. The pseudo-code for this heuristic algorithm is given in Algorithm 3.2.

GPR works as follows. First, the tour $\mathcal{P}$ is initialized (line 1). Next, the cumulative reward for visiting each vertex is calculated and each vertex is marked as feasible (lines 2-5). A vertex may be visited from the right side of the graph (starting from $v_{i,l}$), giving a cumulative reward $R(i,j) = \sum_{n=j}^{l} R(i,n)$, or from the left side of the graph (starting from $v_{i,1}$), giving a cumulative reward $L(i,j) = \sum_{n=1}^{j} R(i,n)$. Note that the cumulative reward is the sum of reward values for all vertices in a row leading to the vertex in question from the appropriate side. Then, the main loop of the algorithm is entered, which runs until all vertices are marked not feasible (line 6). For GPR, a feasible vertex is one that can be reached from the current vertex $\mathcal{P}(end)$ with enough budget left over to return to v_g, closing the tour. Inside the loop, the first thing done is checking if any vertex is not feasible (lines 7-9). This must be done because every iteration the budget changes. Next, heuristics $R'_{i,j}$ and $L'_{i,j}$ are computed for all feasible vertices (lines 10-12). The heuristics are the cumulative

Algorithm 3.2 Greedy Partial-Row Heuristic

```
Input: $AG(w,l)$, $R$, $C$, $B$, $v_s$, $v_g$
Output: $\mathcal{P}$
 1: $\mathcal{P} = \{v_s\}$
 2: for all $v_{i,j} \in V$ do
 3:     $R_{i,j} \leftarrow \sum_{n=j}^{l} R(i,n)$
 4:     $L_{i,j} \leftarrow \sum_{n=1}^{j} R(i,n)$
 5:     $feasible_{i,j} \leftarrow$ True
 6: while $any(feasible) =$ True do
 7:     for all $v_{i,j} \in V$ do
 8:         if $feasible(i,j,\mathcal{P}) \neq$ True then
 9:             $feasible_{i,j} \leftarrow$ False
10:     for all $feasible_{i,j}$ do
11:         $R'_{i,j} \leftarrow R_{i,j}/C(\mathcal{P}(end), v_{i,j}, v_{i,l})$
12:         $L'_{i,j} \leftarrow L_{i,j}/C(\mathcal{P}(end), v_{i,j}, v_{i,1})$
13:     for $i \leftarrow 1$ to $w$ do
14:         if $feasible(i,\mathcal{P}) \neq$ True then
15:             $feasible_i \leftarrow$ False
16:     for all $feasible_i$ do
17:         $R'_{i,1} \leftarrow R_{i,1}/C(\mathcal{P}(end), v_{i,1})$
18:         $L'_{i,l} \leftarrow R_{i,l}/C(\mathcal{P}(end), v_{i,l})$
19:     if $\mathcal{P}(end)_j = l$ then
20:         $best \leftarrow \arg\max R'_{i,1}, R'_{i,j}$
21:         $side = l$
22:     else
23:         $best \leftarrow \arg\max L'_{i,l}, L'_{i,j}$
24:         $side = 1$
25:     to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $best$
26:     if $\mathcal{P}(end)_j \neq 1$ or $l$ then
27:         to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $v_{i,side}$
28:     $feasible_{best} \leftarrow$ False
29:     update $R(i,j)$ and $L(i,j)$ for $i = best_i$
30: to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $v_g$
31: return $\mathcal{P}$
```

reward for each vertex, divided by the cumulative cost to travel to them from either side and then backtrack to exit the row out to the side it was entered from. This is done because the distance between the outside vertex of a row and an inside one is covered twice when a row is partially traversed. Feasibility and heuristic values for full-rows are then calculated in the same manner described for the GR algorithm (lines 16-18). Row heuristics are needed because their computation is fundamentally different since there is no backtracking needed when traversing an entire row. Finally, the loop checks which side of the vineyard the current vertex is on (line 19) and finds the vertex or row with the maximum heuristic value for that side (lines 20 and 23). If a vertex is chosen, the path to that vertex and back to the outside of the row is added to the tour. If a row is chosen, the path to that row and across it to the other side is added to the tour (lines 25-27). The feasibility for all vertices in the newly included path is negated so that these vertices are not considered in future iterations of the loop, and cumulative costs are updated as well (lines 28-29). Lastly, when the main loop has exited, the path from the current position to v_g is added to the tour (line 30).

Like GR, the GPR heuristic algorithm guarantees that the tour will terminate at the correct vertex and the overall used budget will be less than or equal to B. This is because it always checks whether there is enough budget left over to go to the v_g before completing a move. The complexity of GPR is similar to GR, however each iteration of the main loop needs to reevaluate heuristic values for every vertex in the graph, meaning the loop has a complexity of $\mathcal{O}(wl)$, giving the overall algorithm a complexity of $\mathcal{O}(wl + kwl) = \mathcal{O}(w^2l)$. In the case of a square IG, the complexity is $\mathcal{O}(w^3)$, or cubed on the number of rows in the graph.

3.4 Real-World Data Experimentation

To evaluate the efficiency of the algorithms presented in the previous section, an AG was created to represent a real-life vineyard on which the GR and GPR heuristics could be tested. Data was collected in the vineyard to generate reward values representative of the water need for each vine, thereby producing a reward function useful for solving the OP on the constructed graph. GR and GPR were tested against an integer linear program solver and the S-Algorithm for solving the OP. This section describes these experiments, as they were originally discussed in [121].

3.4.1 Vineyard AG and Data Sampling

The AG used in this set of experiments was modeled after a commercial vineyard located near Merced, California. This vineyard consisted of $w = 240$ rows with $l = 500$ grapevines in each row. Each vine is separated by a distance of about 5 feet 6 inches (1.68 meters) and each row is separated by a distance of about 10 feet 6 inches (3.20 meters) for a plot size of roughly 160 acres (64.7 hectare). This translated to a

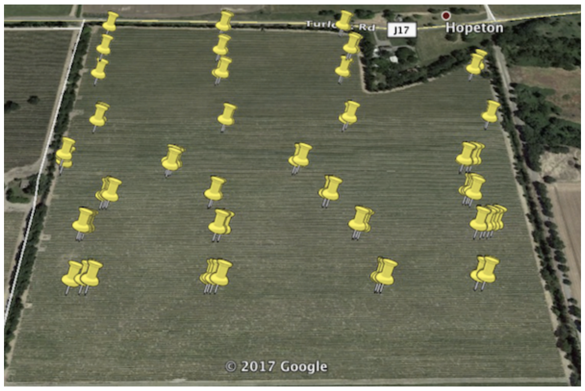


Figure 3.3: Locations shown on a map where soil moisture content was measured in a vineyard. The particular vineyard examined was not rectangular, however for the purposes of this study it was assumed to be.

graph with the same dimensions, and assuming there to be one vertex for every vine, containing $120,000$ vertices.

The main goal of the RAPID project is to use an autonomous robot to adjust irrigation to each vine on an individual basis, and the ensuing OP should have a reward function defining necessary irrigation adjustment at every vertex. Accordingly, this adjustment was inferred by comparing local soil moisture measurements with target values, giving a reward function of $R(v) = |M - m(v)|$ where M is the target moisture value for the vineyard and $m(v)$ is the measured moisture value at vertex v. This function is the absolute difference between the actual amount of water in the soil and the desired level of water in the soil, therefore making the reward a number that shows how under-watered or over-watered a vine is. Here, M is assumed to be a given constant, however in some problems this might not be the case, as some plants may require different target soil moisture values. The measured moisture values $m(v)$ were extracted from a set of sample locations within the vineyard, shown in Figure 3.3, using a Hydrosense HS2P probe manufactured by Campbell Scientific. Sample locations were probed manually within a 2 hour window on a day during the summer growing season, and each location was probed multiple times with the corresponding moisture values averaged. In order to get a value for every vertex in the graph, these samples were interpolated linearly for vertices in between and nearest neighbor was used for vertices that could not be interpolated (which sit between a sample location and the outside of the graph). An example of a reward map created for this particular vineyard is shown in Figure 3.4.

The OP also requires a cost function on the AG. Since vines are spaced equally within rows, the movement cost between them is constant. The chosen robotic platform, a Clearpath Husky, with an average top speed of 3.3 feet per second ($1m/s$), can cover the distance between each vine in 1.68 seconds. The distance between rows is also constant and nearly twice as long, therefore taking 3.2 seconds to traverse each gap. The cost function for the AGOP can be defined in terms of either time

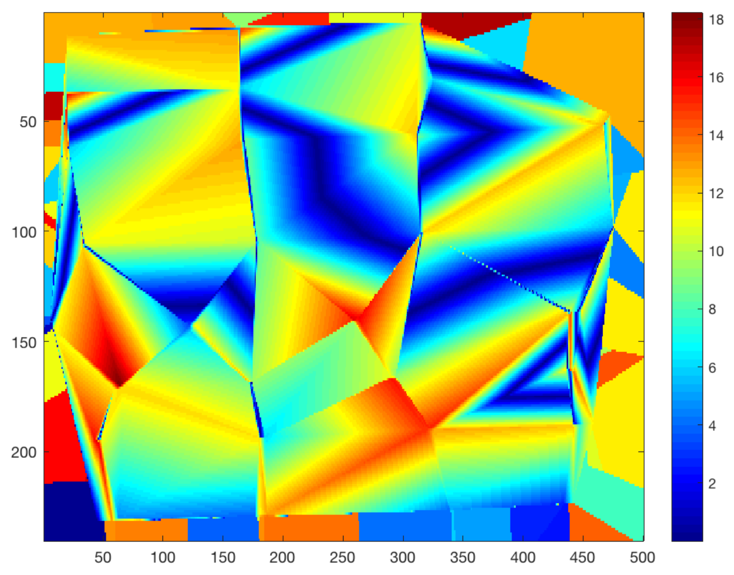


Figure 3.4: The reward map showing the distribution of vertex rewards within the vineyard. Red indicates an area of high reward and blue indicates an area of low reward.

or distance hence either set of values can be used. The budget B should also be defined accordingly. To simplify calculations and representations, the experimental comparisons made in this section normalize edge cost to 1 for both types of edges, turning the problem into a CCAGOP. This does not effect the overall results and the methods presented in section 3.3 can be used on both the CCAGOP and the AGOP.

Lastly, the version of the OP being solved in each experiment is rooted, therefore requiring a start and end vertex. For this particular problem, the solution must be a tour, hence $v_s = v_g$. In each experiment, the root is located in the lower left corner of the vineyard as seen from the aerial view in Figure 3.3.

3.4.2 Method Comparisons

To begin, the first set of experiments performed compared all four methods: the optimal integer linear program, the S-Algorithm, the GR heuristic, and the GPR heuristic. For the optimal method, the solver chosen to compute the solution was the SCIP Optimization Suite [88]. Because of the computational limitations of integer linear programming, the problem size was reduced to an AG with 8 rows and 12 vertices per row, for a total of 96 vertices. This was the largest size graph it was capable of solving the OP for within reasonable time for the experiments performed. In this case, the S-Algorithm was given 3000 trials before returning a solution, and all other parameters were set equal to those suggested in [126]. For each method, the budget was varied to show the capabilities of each across different circumstances.

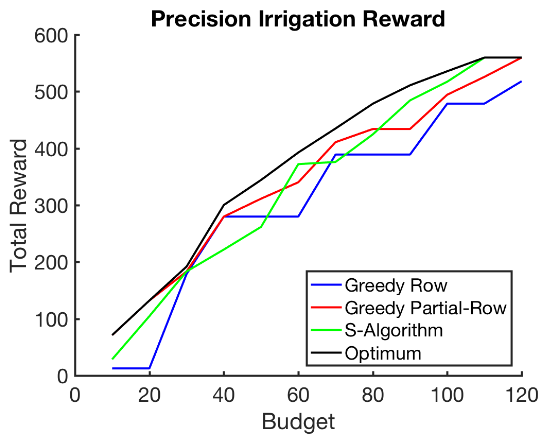


Figure 3.5: Results of each AGOP solution method on the 8×12 graph, showing the amount of reward collected when varying the budget across a range of values.

Results of this first set of experiments is shown in Figure 3.5.

The results show that each of the non-optimal methods is able to compute solutions relatively close to the optimum, suggesting their efficiency in budget utilization. This also hints that the use of approximation algorithms for this type of graph is impractical as the heuristics presented easily overcome the guaranteed $(2 + \varepsilon)$ bound without requiring the affiliated time complexity (see section 2.2 for the related discussion). Through the range of budgets tested, all of the heuristic approaches stay relatively close to each other in terms of reward collected. At a budget of 10, the GPR algorithm matches the optimum, suggesting economical use of small amounts of budget. Each of the algorithms reaches the maximum reward collection, starting with the integer linear program and S-Algorithm at a budget of 110, followed by GPR at 120 and GR at 130.

Figure 3.6 shows the paths produced by each of the AGOP solvers for the case with a budget of 50. The figure exhibits how the results of each algorithm differs. The optimum path contains two fully traversed rows and two partially traversed rows. A similar behavior is shown by GPR, however one of the partial-rows was incorrect. This observation demonstrates that GPR produces highly effective solutions. GR, on the other hand, was not designed to accommodate these and therefore suffered in overall reward collected even though the full-rows it added to the path were the same ones used in the optimal path. The S-Algorithm displays the ability to take partial-rows as part of its path, however many of the vertices visited (including both partial-rows and full-rows) were not the same as in the optimal path.

The second set of experiments performed compared the three heuristic methods against each other without the integer linear program method. For this set of experiments, the size of the AG was changed to 60 rows with 60 vertices per row (60×60) for a total of 3600 vertices. The full-sized graph matching the size of the vineyard was again not used because the S-Algorithm could not compute a solution on such a large graph in sufficient time. Additionally, the number of Monte Carlo trials the

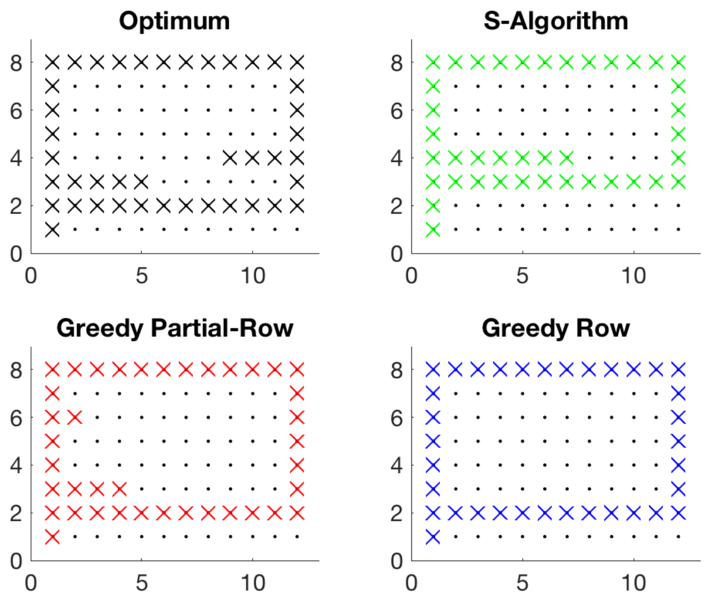


Figure 3.6: A comparison between the paths created by each of the solution methods on an 8×12 AG with $B = 50$. In this case, the starting vertex is the bottom left, $v_{1,1}$).

S-Algorithm was given was reduced to 500 for the same reason. Other parameters stayed the same. Results for this second set of experiments are shown in Figure 3.7. These experiments show a clear performance gap between each method, with GPR gathering the most reward within the given problems budget. The gap between GPR and GR is significant, showcasing how useful the ability to traverse partial rows is. The gap between GPR and the S-Algorthm is less pronounced, and would likely be smaller if the S-Algorithm were allowed more trials to find a better solution. However increasing the number of trial would not be a fair comparison, as the S-Algorithm requires significantly more time to produce a solution because its computational complexity depends on the number of trials is is allowed to compute.

The third and final set of experiments performed compared only GPR and GR on the full-sized 240×500 graph with $120,000$ vertices. Again, only these two methods were used on the full-sized AG because only they were capable of finding solutions on them in reasonable time (for perspective, the S-Algorithm would not find a single solution after 1 week of computation). The results of these experiments are shown in Figure 3.8. From Figure 3.8a, it is clear that GPR performs better than GR on this size of graph as well. Figure 3.8b shows how much residual budget is leftover after completing a tour, across the same range of budgets. The residual budget shows how wasteful an algorithm is, and this makes it clear that GR is extremely wasteful with its budget, hence why it is less efficient at collecting rewards. This makes sense, as GR is unable to utilize any amount of leftover budget smaller than the cost of

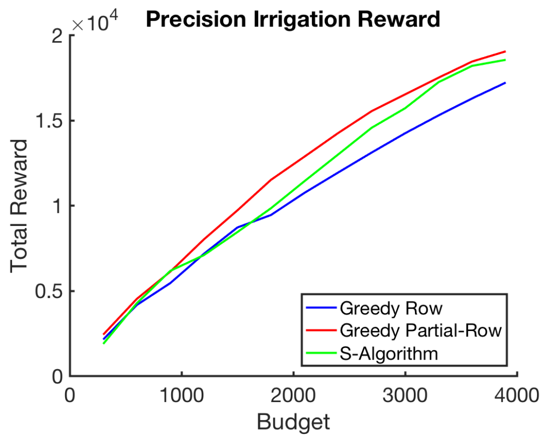


Figure 3.7: Results of the three heuristic AGOP solution methods on a 60×60 graph, showing the amount of reward collected when varying the budget across a range of values.

traversing a full-row twice, whereas GPR can use the leftover to partially traverse a row. Overall, these factors demonstrate that the GPR heuristic algorithm is superior for use on AGs.

3.5 Conclusion

In this chapter a version of the OP was studied where routing was required on a special class of graphs that are arranged in an aisle format. This emerges as a problem for use on robots deployed for precision irrigation where motions are constrained by the rows of a vineyard. The problem was proven to remain NP-hard on this specific type of graph and two heuristics, GR and GPR, were presented to provide good solutions on very large graph sizes. These were compared against an optimal solution using an integer linear program formulation and a commonly used heuristic called the S-Algorithm on graphs much smaller than necessary for the domain of precision agriculture. Of them, the GPR heuristic performs nearly as well as the optimal solution and is able to quickly compute routes on very large graphs consisting of over $100,000$ vertices. This shows that domain specific heuristics are very efficient and very practical for specific use cases such as the one presented in Chapter 1.

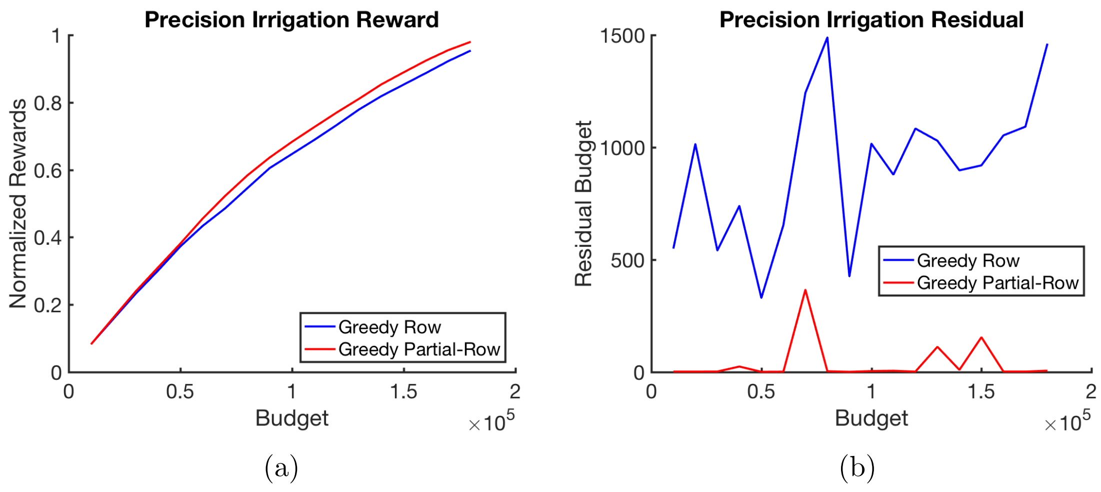


Figure 3.8: (a) Results of the GPR and GR heuristic methods on a 240 × 500 graph, showing the amount of reward collected when varying the budget across a range of values. The y-axis shows normalized reward values, or collected reward divided by the total reward for all vertices in the graph. (b) A plot showing the unused or residual budget after completing a tour for the GPR and GR heuristics.

Chapter 4

Multi-Robot Orienteering in Vineyards

This chapter extends the problem of orienteering in vineyards to the use case with a team of robots working together. Heuristic algorithms are developed, based on work originally presented in [120, 123].

4.1 TOP Background

In this section, the TOP is formulated and two solution methods for the general case problem are given, one based on integer linear programming and one heuristic method. The heuristic method, GLS, is later used as a benchmark for performance comparison against the methods presented later in this chapter.

4.1.1 Problem Definition

The TOP is defined similarly to the OP, with the key difference of utilizing multiple agents to collect rewards. Each agent a follows its own path $\mathcal{P}_a \subset V$ in the given graph $G(V, E)$, visiting some ordered subset of vertices $\mathcal{P}_a(i) = v \in V$ and edges $\mathcal{P}_{e,a}(i) = e \in E$, where $1 \leq i \leq |\mathcal{P}|$. For the version of the TOP focused on here, the start vertex v_s and goal vertex v_g are the same for each agent, i.e. $\mathcal{P}_a(1) = v_s \forall a$ and $\mathcal{P}_a(|\mathcal{P}_a|) = v_g$, though generally this is not necessary. Indeed, it is common to have multiple start/goal locations to act as "depots" for one or more agents. It is also assumed that the reward and cost functions for each agent are identical, such that every agent receives the same amount of reward for visiting a vertex $r_i = R(v_i)$ and incurs the same amount of cost for traversing an edge $c_{i,j} = C(e_{i,j})$. In some versions of the problem, this is not the case, as these have agents that are non-homogeneous. While useful for modeling complex problems, this generalization is not used here. Similarly, a general case TOP will have a budget B_a for each agent which may or may not be the same, whereas here the problem is defined with a given budget B that is the same for all agents (each agent still has its own separate budget). Thus, this describes the case where all agents involved in the TOP are homogeneous.

Given a graph $G(V, E)$ with a reward function on each vertex $r : V \to \mathbb{R}_{\geq 0}$, a cost function on each edge $c : V \to \mathbb{R}_{\geq 0}$, a start vertex $v_s \in V$, a goal vertex $v_g \in V$, a budget $B \in \mathbb{R}_{\geq 0}$, and a set of agents A, the TOP asks to find $|A|$ paths $\mathcal{P}_a$ that start at v_s and end at v_g, each having a cumulative cost no more than B, that maximizes the sum of collected rewards across all paths. As in the single agent OP, a vertex can only contribute its reward once. If any vertex is visited more than once, whether in a single path or in multiple paths, its reward is added only on the first visit and all other visits obtain 0 reward. Likewise, an edge contributes to the total cost of an agent's path every time that agent traverses that edge, regardless of how many times that edge has been crossed by any agent. These facts are important when considering graphs that are not complete, however any graph for which a TOP exists as characterized here can be made complete by introducing edges of cost equal to the shortest path between the vertices they connect. Finally, because the OP is a special case of the TOP (where $|A| = 1$), the TOP is obviously NP-hard as well.

4.1.2 TOP Linear Programming

Like the OP, the TOP can be formulated as an integer linear program which can then be solved to produce an optimal solution. The version of this formulation shown here is taken from [130], rewritten with the notation used in this dissertation. There are three types of decision variables used in the team problem, $x_{i,j,a}$ describing the usage of edge $e_{i,j}$ in a path by agent a (1 if it is a part of $\mathcal{P}_a$ and 0 otherwise), $u_{i,a}$ indicating the position of vertex v_i in path $\mathcal{P}_a$ (such that $\mathcal{P}_a(u_i - 1) = v_i$), and $y_{i,a}$ specifying if the reward for vertex v_i is collected by $\mathcal{P}_a$ (1 if so and 0 otherwise).

$$\max \sum_{a\in A}\sum_{v_i\in V} r_i y_{i,a} \tag{4.1}$$

$$s.t. \sum_{v_i\in V}\sum_{v_j\in V} c_{i,j}x_{i,j,a} \leq B \qquad \forall a \in A \tag{4.2}$$

$$\sum_{v_i\in V} x_{i,k,a} = \sum_{v_j\in V} x_{k,j,a} = y_{k,a} \qquad \forall k \in V; \forall a \in A \tag{4.3}$$

$$\sum_{a\in A} y_{k,a} \leq 1 \qquad \forall k \in V\backslash\{v_s, v_g\} \tag{4.4}$$

$$2 \leq u_{k,a} \leq |V| \qquad \forall v_k \in V\backslash\{v_s\}; \forall a \in A \tag{4.5}$$

$$u_{i,a} - u_{j,a} + 1 \leq (|V| - 1)(1 - x_{i,j,a}) \qquad \forall v_i, v_j \in V\backslash\{v_s\}; \forall a \in A \tag{4.6}$$

$$\sum_{a\in A}\sum_{v_j\in V\backslash\{v_s\}} x_{v_s,j,a} = \sum_{a\in A}\sum_{v_i\in V\backslash\{v_g\}} x_{i,v_g,a} = |A| \tag{4.7}$$

The linear program for the TOP is very similar to the one for the OP given in section 3.1. It can be understood as follows: Equation 4.1 is the objective function, stating that the total reward collected across all agents for each vertex in the resulting paths is maximized. Equation 4.2 is the budget constraint, enforcing that every agent has a bounded path cost less than the budget, i.e. $\sum_{i=1}^{|\mathcal{P}_a|-1} C(\mathcal{P}_{e,a}(i)) \leq B$. Constraint Equation 4.4 requires that each vertex aside from v_s and v_g is only visited once (this assumes a complete graph). The constraint in Equation 4.3 requires that each edge can be included in some agent's path only once (again, this is assuming the use of a complete graph). The two Miller-Tucker-Zemlin subtour elimination constraints are given by Equation 4.5 and Equation 4.6, which certify that each agent has a single continuous path. Finally, Equation 4.7 requires every agent's path start at v_s and end at v_g. In the unrooted TOP, this last constraint can be removed. When the solution of the integer linear program is returned, the set of paths can be extracted from the use of edges given by $x_{i,j,a}$ with the sequence of vertices specified by $u_{i,a}$ for each agent.

4.1.3 Guided Local Search Heuristic

The TOP, being NP-hard, necessarily requires heuristic methods to find solutions to problems with a large number of agents and on graphs with a large number of vertices. One of the most common varieties of heuristic used is the multi-phase meta-heuristic, which makes use of many types of path improvement heuristics combining them into a single procedural algorithm for the TOP. The GLS meta-heuristic, given in [128], is one of these algorithms. Here, GLS is briefly described to be used later as a performance benchmark, and was chosen because it is easily extendable from the general case TOP to the case of AG. It works on graphs that are fully connected as well as graphs that are not, befitting of AG (though without loss of generality an AG can be made fully connected by supplementing shortest path edges).

GLS begins in a construction phase, creating an initial set of $|A|$ paths starting at the start vertex v_s and ending at the goal vertex v_g. Each of these paths $\mathcal{P}_a$ is connected to only one other vertex (which is unique for each path) that is far from both v_s and v_g. Cheapest insertion is performed on each tour until no more insertions can be made to increase the number of vertices visited before starting the main loop of the algorithm. The main loop continues until a suitable solution is found or until a max number of iterations has occurred. A suitable solution is one which can no longer be improved for the defined parameters, and this is selected as the best solution that has already been "disturbed", which is a process that removes a fixed percentage of vertices from each path to allow opportunity for a better solution to be found. At each iteration, if the current solution is the best one found, then it is saved and then disturbed. Likewise, if it is not the best one found, then it can still be disturbed if some criteria is met.

Inside of the main loop is another loop that uses a number of local search heuristics to improve the current solution: "swap" trades vertices in one agent's path for vertices in another, "TSP" performs a 2-opt edge swap on individual paths to decrease total cost, "move" takes a group of vertices from one agent's path and moves them to another agent's path, "insert" adds vertices that are not visited by any agent to a path, and "replace" swaps a visited vertex in a path for an unvisited one. Inside of the "replace" and "TSP" heuristics, the algorithm uses guided local searches (hence the name) to improve on their performance.

GLS requires numerous parameters as inputs that should be tuned for good performance, some of which do not intuitively increase reward collection. Additionally, GLS is rather unsuited for large problem sizes of more than $1,000$ vertices (though this is true for most TOP heuristic methods). The complexity of GLS is difficult to analyze due to how the parameters interact, and the authors of [128] do not attempt to, however the best estimate from [123] is $\mathcal{O}(B \cdot |V|^2 log^2(|A| \cdot |V|))$ where B is the budget, $|V|$ is the number of vertices in the graph, and $|A|$ is the number of agents. Note the linear dependency on B.

4.2 Team Orienteering on Aisle Graphs

Much like section 3.2, this section focuses on developing insights for solving the TOP on AGs. As before, little or no work has been done using this type of graph structure in the TOP. The insights presented were originally developed in [120].

4.2.1 Team Coordination of Robots

In the single agent OP, it is not necessary to coordinate movement as the are no other agents to coordinate with. In the multi-agent TOP, the question arises whether or not all paths chosen by the solver can be traversed concurrently by the agents. It is a typical assumption that every agent will begin traversing its path at the same time (though this is not always the case, and must be specified), thus it is necessary to ensure that no conflicts arise. These conflicts are emergent from the particular problem being solved, most commonly being the inability to share edges, and should be embedded in the problem's graph. An intuitive way to do this is to discourage usage of shared edges by changing the cost of an edge to ∞ after it has been included in one agent's path. However this can be very inefficient, because it completely locks out other agents from using that edge. In a fully connected graph, this should not be of any concern, since the optimal solution would not utilize any edge more than once. Most real world problems, such as vineyard orienteering, do not have fully connected graphs and the graphs are only made to be in preprocessing. The composite edges are paths of minimal cost connecting two vertices which will necessarily avoid the infinite cost edges. In these cases, edges with infinite cost are extremely limiting because they force agents to take roundabout paths rather than taking a direct route. With enough edges of infinite cost, some vertices may become entirely unreachable. Therefore, a more advanced technique is required for obtaining good solutions on team cooperation problems.

To counter the problems presented by making edges traversed by other agents have costs equal to ∞, it is possible to limit when edges have infinite costs. To do this, the cost function should be augmented to be dependent on spent time. Here, "time" refers to cost, considering that they are functionally equivalent when using homogeneous agents as mentioned in section 4.1. The times during which an edge is used should be kept track of using a new boolean function $T_{0,1}(e, t) \rightarrow \{0, 1\}$ which returns 1 if the edge e is in use at time t and 0 if it is not. Here, time t is the accumulated cost since the beginning of a path. The new cost function is given as a piece-wise linear function:

$$T(e,t) = \begin{cases} \infty & \text{if } T_{0,1}(e,t) = 1 \\ C(e) & \text{otherwise} \end{cases} \tag{4.8}$$

Now the total cost of a path should be defined according to the new cost function:

$$T(\mathcal{P}, t_0) = T(\mathcal{P}_e(1), t_0) + \sum_{i=2}^{|\mathcal{P}|-1} T(\mathcal{P}_e(i), T(\mathcal{P}(1, i-1), t_0) + t_0) \tag{4.9}$$

The new cost function assumes an agent begins traversing the path $\mathcal{P}$ at time t_0. Note that the total cost of a path is a cumulative function, which must evaluate $T_{0,1}$ for every edge in the path at the current time. With the assumption that all agents start traversing their paths at $t_0 = 0$, notation is simplified, e.g. $T(\mathcal{P}) \equiv T(\mathcal{P}, 0)$.

4.2.2 AG Movement Restrictions

Having defined how agents must coordinate movement over a graph, it is now possible to specify this behavior on AGs. Since AGs are often modeled after real world environments, such as vineyards or warehouses, the limitations placed on robot movement are physical and easily demonstrated. For example, in Figure 4.1 a Clearpath Husky robot is shown facing the rows of a vineyard. This illustrates the small amount of space within each row of a vineyard for maneuvering. In a scenario where two or more robots are navigating the same row at the same time and must cross each other's path, it is entirely possible or perhaps very likely that they will collide. This event can be calamitous and cause catastrophic failure of either or both robots. Therefore it is imperative that this situation is avoided at all costs (justifying the use of occupied edge costs of ∞). To successfully avoid collisions, robots should avoid entering into rows that are already occupied by other robots. Or, more specifically, robots should avoid traversing portions of rows in which another robot is operating or about to operate. This makes it critical for each robot to know every other robot's planned path, to avoid crossing paths.

While the insides of rows may be tight, the outer columns usually are not. Indeed, Figure 4.1 shows this is true for vineyards. In vineyards, the outer columns separate individual vineyard blocks from each other or from other structures, and are typically built large enough to accommodate multiple tractors, trucks, or worker vehicles. Thus the movement constraint limiting only a single robot does not apply. Multiple robots can safely pass each other in these outer parts of the vineyard without chance of collision (assuming properly working obstacle avoidance). This low risk environment on the outskirts of rows usually also exists in other structures of similar arrangement, such as warehouses, and can be thought of as a general property of AGs.

These two very different constraints for the driveability of robots on AGs mean that special care must be taken to define the appropriate cost functions on these

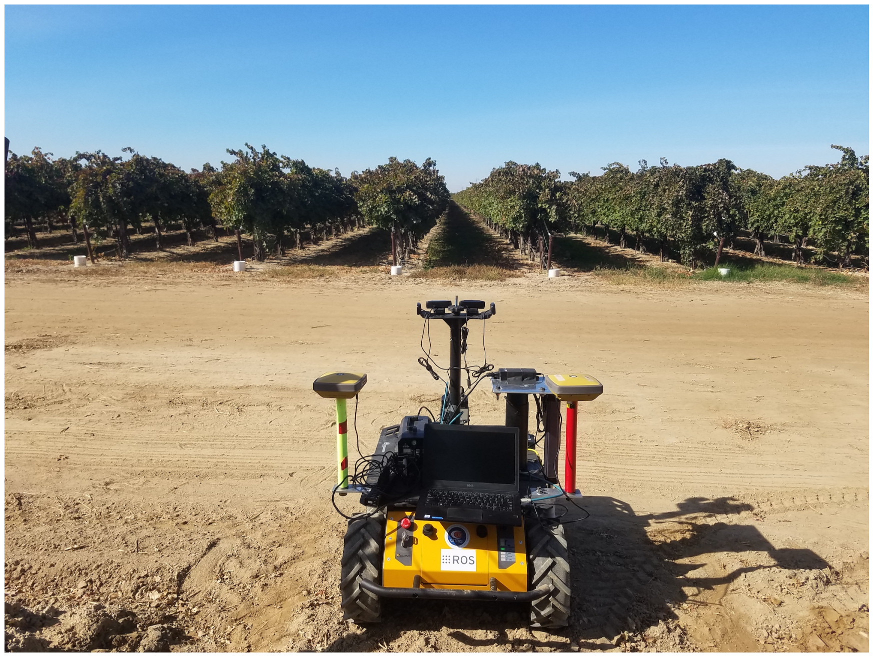


Figure 4.1: A Clearpath Husky robot next to a vineyard located in Central California. The distance between rows is clearly visible and demonstrates how little room there is for maneuverability of multiple robots.

graphs. The set of edges can be split into two subsets, the set of outer edges $E_o \subset E$ and the set of row edges $E_r \subset E$. Outer edges are not limiting in the sense of multi-robot restrictions, so the cost function of these edges remains the same $t = C(e)$ for all $e \in E_o$. Notice here that t is used instead of c as in section 3.1. Row edges do have restrictions, so the cost function of these edges changes to that described earlier this section, i.e. $t = T(e, t_0)$ for all $e \in E_r$. Now it is possible to define two special versions of the TOP on AGs:

Constant Cost Aisle Graph Team Orienteering Problem (CCAGTOP): Given a graph $G(V, E) = AG(w, l)$ with constant cost function $C(e) = k$; $\forall e \in E_o \subset E$, piece-wise cost function $T(e, t_0)$; $\forall e \in E_r \subset E$, reward function R, vertices within the graph $v_s, v_g \in V$, and constants B, t_0, find a path $\mathcal{P} \subset V$ starting at time t_0 with cost no more than B that begins at v_s and ends at v_g which maximizes the cumulative reward of visited vertices.

Aisle Graph Team Orienteering Problem (AGTOP): Given a graph $G(V, E) = AG(w, l)$ with cost function $C(e)$; $\forall e \in E_o \subset E$, piece-wise cost function $T(e, t_0)$; $\forall e \in E_r \subset E$, reward function R, vertices within the graph $v_s, v_g \in V$, and constants B, t_0, find a path $\mathcal{P} \subset V$ starting at time t_0 with cost no more than B that begins at v_s and ends at v_g which maximizes the cumulative reward of visited vertices.

In the former [120, 123], the AGTOP was known as the IGTOP. These are team versions of the CCAGOP and AGOP, respectively, and thus are both NP-hard, since the CCAGOP and AGOP are special cases of these problems with only a single agent.

4.3 Extending GPR to the AGTOP

The vast size of real-life vineyards mean that a single battery powered robot may not be able to cover much ground by itself. Instead, a team of robots may be needed to provide adequate coverage of the vineyard. Again, heuristics that can be scaled to tackle problems of large size are needed. The low complexity and effective results displayed by the GPR heuristic algorithm in Chapter 3 for a single agent motivate its extension into the domain of multi-agent heuristics. The following algorithms for solving the AGTOP, taken from [120, 123], are all based on GPR. As before, the start and goal vertices are the same and the path produced is a tour.

4.3.1 Vineyard Sectioning

A straightforward and uncomplicated method of solving the AGTOP and building paths for multiple agents on a single AG is to use a divide-and-conquer approach. This involves splitting the graph into multiple parts and solving the AGOP on each. Therefore, for a set of agents A, the AG is split into $|A|$ sections and GPR is run on each section. A simple algorithm that does this while trying to normalize the potential collected reward for each agent, called Vineyard Sectioning (VS), is presented in Algorithm 4.3.

Algorithm 4.3 Vineyard Sectioning

```
Input: $AG(w,l)$, $R$, $C$, $B$, $v_s$, $v_g$, $|A|$
Output: $\mathcal{P}_1 \dots \mathcal{P}_{|A|}$
 1: $R_{AG} = \sum_{v \in V} R(v)$
 2: $i = 0$
 3: for $a$ to $|A|$ do
 4:     $k = i$
 5:     $y = 0$
 6:     while $y/R_{AG} < B/|A|$ do
 7:         $i = i + 1$
 8:         $y = y + \sum_{j=1}^{l} R(i,j)$
 9:     $R_a(i,j) = R(i,j)$; $\forall i = 1 \dots w, j = 1 \dots l$
10:     for $0 < z < k$ and $i < z \leq w$ do
11:         $R_a(z,j) = 0$ for $j = 1 \dots l$
12:     $\mathcal{P}_a \leftarrow$ GPR($AG(w,l)$, $R_a$, $C$, $B$, $v_s$, $v_g$)
13: return $\mathcal{P}_1 \dots \mathcal{P}_{|A|}$
```

VS works as follows: First, the total reward of all vertices in the graph is computed (line 1). Next, a variable i, which holds the index number of the current row that the algorithm is working on, is initialized (line 2). Then, the main loop of the algorithm begins, which iterates over all the agents that need a path (line 3). A temporary variable k is set to hold the index number of the row that begins the current section of the graph (line 4). Then, a loop iteratively adds rows to the current section, which ends at row i, until the reward of the section is roughly equal to the total reward of the AG divided by the number of agents (lines 6-8). If the budget is not the same for each agent, then the section should hold a proportional amount of reward to better utilize the budgets of each agent. Next, a temporary reward function is made from the original that has zero reward for all vertices in rows outside of the current section between rows k and i (lines 9-11). Finally, GPR is run on this temporary graph and a path is created for the current agent (line 12).

At the end of VS, there will be $|A|$ paths built as tours, none of which can cause collisions between the agents. This is because all the sections of the vineyard are disjoint, and GPR will never build a path containing a row of an AG with zero reward, so there will never be any overlapping sections. The rows visited by one agent will never be visited by another agent. The complexity of this algorithm is $\mathcal{O}(|A| \cdot w^2 l)$ because GPR is run for each agent once.

4.3.2 Series GPR

Another method of extending GPR to the multi-agent case is to run many instances of GPR in succession to produce multiple paths. This approach, called Series GPR (SGPR), has the advantage of allowing every agent to consider the entirety of the AG without restricting its movement to only a subsection of the graph. It works by running a modified version of GPR $|A|$ times in succession, setting the reward value for vertices visited in previous runs to zero before constructing the next path, meaning each new path will attempt to explore previously unexplored high value areas. However, GPR must be modified to prevent potential collisions that arise when multiple agents travel within the same row at the same time, as mentioned in section 4.2. This modification allows the agent to wait at a location until the row is opened and no threat of collision occurs. SGPR is presented in algorithm Algorithm 4.4 and the modified GPR with Avoidance (GPRA), is presented in algorithm Algorithm 4.5.

Algorithm 4.4 Series GPR

```
Input: $AG(w,l)$, $R$, $C$, $B$, $v_s$, $v_g$
Output: $\mathcal{P}_1 \dots \mathcal{P}_{|A|}$, $\mathcal{TM}$
1: initialize $\mathcal{TM}$
2: for $|A|$ agents do
3:     $\mathcal{P}_a$,$\mathcal{CM} \leftarrow$ GPRA($AG(w,l)$, $R$, $C$, $B$, $v_s$, $v_g$, $\mathcal{TM}$)
4:     for all $v \in \mathcal{P}_a$ do
5:         $R(v) = 0$
6: return $\mathcal{P}_1 \dots \mathcal{P}_{|A|}$, $\mathcal{TM}$
```

Algorithm 4.5 GPR with Avoidance

```
Input: $AG(w, l)$, $R$, $C$, $B$, $v_s$, $v_g$, $\mathcal{TM}$
Output: $\mathcal{P}$
 1: $\mathcal{P} \leftarrow \emptyset$
 2: $t_{wait} = 0$
 3: Compute cumulative rewards from left $L_{i,j}$ and right $R_{i,j}$
 4: while $\mathcal{P}(end) \neq v_g$ do
 5:     Reset $\mathcal{TM}$ to input
 6:     Set vertex feasibility $feasible_{i,j} \leftarrow$ True for all vertices
 7:     $\mathcal{P} = \{v_s\}$
 8:     while $any(feasible) =$ True do
 9:         Compute heuristics of full-row and partial-row for current side
10:         Compute times for crossing edges in each full-row $t_{fr}$ and partial-row $t_{pr}$
11:         Find best full-row $best_{fr}$ without time conflicts in $\mathcal{TM}(t_{fr})$
12:         Find best partial-row $best_{pr}$ without time conflicts in $\mathcal{TM}(t_{pr})$
13:         if $best_{fr} \neq \emptyset$ or $best_{pr} \neq \emptyset$ then
14:             if $feasible(best_{fr}) =$ False and $feasible(best_{pr}) =$ False then
15:                 Mark as not feasible
16:             if $R(best_{fr}) >= R(best_{pr})$ and $feasible(best_{fr}) =$ True then
17:                 Append $best_{fr}$ to $\mathcal{P}$
18:                 Add vertices in $best_{fr}$ to $\mathcal{TM}(t_{fr})$
19:                 Mark $best_{fr}$ as not feasible
20:             else if $feasible(best_{fr}) =$ True then
21:                 Append $best_{pr}$ to $\mathcal{P}$
22:                 Add vertices in $best_{pr}$ to $\mathcal{TM}(t_{pr})$
23:                 Mark $best_{pr}$ as not feasible
24:         else
25:             Tell agent to wait $waittime$
26:     Tell agent to wait $t_{wait}$
27:     if exists path from $\mathcal{P}(end)$ to $v_g$ without conifilct in $\mathcal{TM}$ then
28:         Add vertices and times to $\mathcal{TM}(T(\mathcal{P}))$
29:         Append path to $\mathcal{P}$
30:         return $\mathcal{P}$, $\mathcal{TM}$
31:     else
32:         $t_{wait} = t_{wait} + waittime$
```

SGPR works as follows. First, a time map $\mathcal{TM}$ that holds the vertices at which each robot might be at any given time is initiallized (line 1). Then, a loop that iterates for each agent is entered (line 2). GPRA is run for the current agent using the most recent time map and reward function, and outputs a collision-free path for the agent plus the time map augmented with new vertices and times (line 3). Next, all visited vertices have their rewards set to zero in the reward function and the loop iterates for the next agent (lines 4-5).

GPRA works similarly to GPR but with a few modifications. The entire algorithm is wrapped in a while loop that checks if a valid path is completed (line 4). A path is valid if it ends at the goal vertex v_g within the budget B. Inside the main loop, the time that each vertex will be visited is computed in addition to the full-row and partial-row heuristics (lines 9-10). The best full-row and partial-row without time conflicts are checked for feasibility and, if feasible, the better of the two is chosen for inclusion in $\mathcal{P}$ (lines 11-23). Note that if the heuristic values are equal, the full-row will be added to the path. All newly visited vertices and times are added to $\mathcal{TM}$ and marked not feasible (lines 17-19, 21-23). If no full-rows or partial-rows without time conflicts exist, then the agent is told to wait one time unit (line 25), where *waittime* is a given constant. Once there are no more feasible vertices, the route to the goal vertex v_g is then appended to the path after having waited an appropriate amount t_{wait} (lines 26-29). If a time conflict occurs when building this last portion of the path, from $\mathcal{P}(end)$ to v_g, then extra wait time t_{wait} is need for the end, and the path must be rebuilt with that contingency (lines 31-32). Eventually, the algorithm returns a valid path $\mathcal{P}$ and time map $\mathcal{TM}$ which is used to build more paths (line 30).

Since GPRA is an augmented version of GPR, it still provides the same guarantees of terminating at the correct vertex and ending within the budget. However, it also provides conflict avoidance functionality, taking into account where other agents are at all times. GPRA has a computational complexity of $\mathcal{O}(w^2 l)$, therefore giving SGPR an overall complexity of $\mathcal{O}(|A| \cdot w^2 l)$.

4.3.3 Parallel GPR

The final and most complex method of extending GPR to work with multiple agents is the Parallel GPR (PGPR) algorithm. Like SGPR, all agents are allowed to explore the whole graph, and collisions are avoided using a time map. However, instead of building paths one after the other in serial, all paths are constructed together in parallel. This allows agents to maximize their efficiency by considering heuristics of rows and partial rows for every agent before adding one to an agent's path. Pseudo-code for PGPR is shown in algorithm 4.6.

PGPR works as follows. First, cumulative rewards from both sides for each vertex are computed (line 2). Then, a loop begins which tracks if every path is valid (line 4). If the all paths reach the goal vertex within B, the algorithm returns the set of paths and the updated time map (lines 36-37). However, if one of the paths is not valid, then the invalid path is forced to save more time for the end so it has room

Algorithm 4.6 Parallel GPR

```
Input: $AG(w,l)$, $R$, $C$, $B$, $v_s$, $v_g$, $\mathcal{TM}$
Output: $\mathcal{P}_1 \dots \mathcal{P}_{|A|}$, $\mathcal{TM}$
 1: $\mathcal{P}_a \leftarrow \emptyset$ for $i = 1 \dots |A|$
 2: Compute cumulative rewards from left $L_{i,j}$ and right $R_{i,j}$
 3: $t^a_{wait} = 0$ for $a = 1 \dots |A|$
 4: while any $\mathcal{P}_a(end) \neq v_g$ do
 5:     Reset $\mathcal{TM}$ and vertex feasibility, initialize $\mathcal{BL} = \emptyset$
 6:     $\mathcal{P}_a \leftarrow v_s$ for $i = 1 \dots |A|$
 7:     while $any(feasible) =$ True do
 8:         Clear $\mathcal{CL}$
 9:         for $a = 1 \dots |A|$ do
10:             Compute heuristics of agent $a$ of full-row and partial-row for current side
11:             Compute times for agent $a$ to cross edges in each full-row $t_{fr}$ and partial-row
                $t_{pr}$
12:             for all full-rows not in $\mathcal{TM}$ do
13:                 Find best full-row $best_{fr}$ not in $\mathcal{BL}$ for agent $a$
14:                 Add $best_{fr}$ to $\mathcal{CL}$
15:             for all partial-rows not in $\mathcal{TM}$ do
16:                 Find best partial-row $best_{pr}$ not in $\mathcal{BL}$ for agent $a$
17:                 Add $best_{pr}$ to $\mathcal{CL}$
18:             if nothing added to $\mathcal{CL}$ for agent $a$ then
19:                 Tell agent $a$ to wait $waittime$
20:         while $\mathcal{CL} \neq \emptyset$ do
21:             Search $\mathcal{CL}$ for candidate with highest heuristic value $best$
22:             if $feasible(best) =$ True then
23:                 Append $best$ to $\mathcal{P}_a$
24:                 Add vertices and times to $\mathcal{TM}(T(\mathcal{P}_a))$
25:                 Mark $best$ as not feasible
26:             else
27:                 Add $best$ to $\mathcal{BL}$
28:         Mark vertices in $\mathcal{BL}$ for all agents as not feasible
29:     for all $a = 1 \dots |A|$ do
30:         tell agent $a$ to wait $t^a_{wait}$
31:         if exists path from $\mathcal{P}_a(end)$ to $v_g$ without conflict in $\mathcal{TM}$ then
32:             Add vertices and times to $\mathcal{TM}(T(\mathcal{P}_a))$
33:             Append path to $\mathcal{P}_a$
34:         else
35:             $t^a_{wait} = t^a_{wait} + waittime$
36:     if $\mathcal{P}_a(end) = v_g$ forall $a = 1 \dots |A|$ then
37:         return $\mathcal{P}_1 \dots \mathcal{P}_{|A|}$, $\mathcal{TM}$
```

in its budget to wait for an opening valid path to v_g (line 30,35) and all paths are recomputed by restarting the main loop. Inside this loop, the time map $\mathcal{TM}$ is reset at each iteration, all vertices are marked as feasible, an empty blacklist $\mathcal{BL}$ is initialized, and all paths are cleared (lines 5-6). Next, another loop starts and runs until there are no more feasible vertices in the graph (line 7). Here, feasibility means that any one of the agents can visit the vertex with enough leftover budget to get to v_g. Inside the inner loop, a list of candidate routes is kept, which holds all full-rows and partial-rows that are in consideration for adding to a path next, and this list is cleared at the beginning of every iteration (line 8). For each agent, full-row and partial row heuristics are computed along with visiting times, then the best full-row and partial-row without time conflicts are added to the list of candidates (lines 10-17). If an agent has no candidates, then that agent is told to wait *waittime* (lines 18-19). Next, the list of candidates is iterated over, such that the candidate route with the greatest heuristic value is added to the appropriate path, updating the time map and vertex feasibility, until no more candidates are left (lines 20-27). If any candidate routes conflict with each other, then the lesser candidates will not be feasible and its vertices will be added to $\mathcal{BL}$ for that agent (line 27). Any vertex that is blacklisted from all agents is marked as not feasible (line 28). After all vertices in the graph are marked as not feasible, the route from the current vertex of each path to v_g is appended, as long as these routes do not have conflicts (lines 29-35).

This algorithm completes valid paths for every agent, and does so while avoiding collisions. It works similarly to SGPR, but instead the loop that handles multiple agents is moved from the outside of the main loop to the inside. Thus, the complexity does not change; PGPR works in $\mathcal{O}(|A| \cdot w^2 l)$ time.

4.4 Experimental Comparison

The algorithms presented in the preceding sections were compared on AGs created using a real-life vineyard as a graph model with soil moisture data used to compute reward for each vertex. section 3.4 explains the vineyard and data set used. By running each of the AGTOP methods on this model, a fair evaluation of the efficiency of each was constructed. Variables considered consisted of the size of the graph which was reduced in size to allow the use of the GLS method, the budget of each agent, and the number agents acting as a team.

4.4.1 Team GPR Methods Versus GLS

For the first set of tests, the three AGTOP algorithms (VS, SGPR, and PGPR) were studied in relation to the GLS meta-heuristic method discussed in section 4.1. For these tests, the AG was scaled down to a size of $w = 12$ rows with $l = 25$ vertices per row, for a total of 300 vertices. This was done because of the computational constraints of using GLS on larger graphs. Through all the tests done, no collisions occurred between the paths for any agent. While expected from the AGTOP methods,

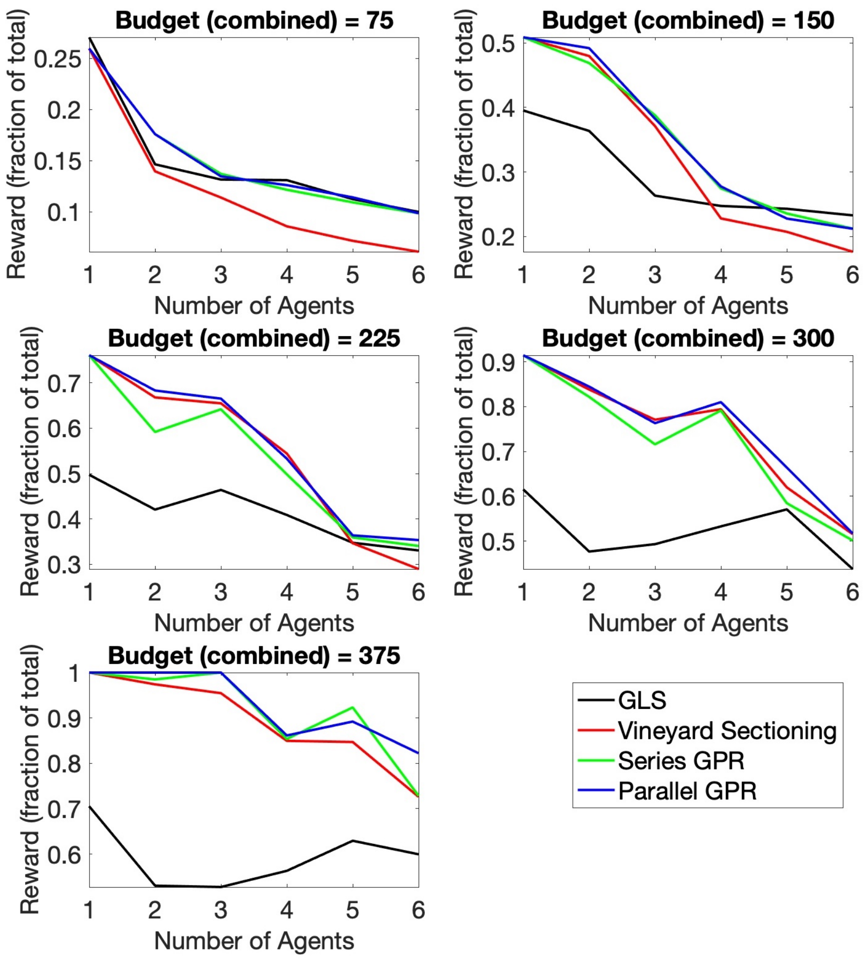


Figure 4.2: The fraction of available reward collected by a number of agents for a given fixed budget on a 12 × 25 graph. Note there are some overlapping lines where values are the same between multiple algorithms.

this is a somewhat unexpected result for GLS, as it does not contain any in-built coordination between agents to prevent occupying edges simultaneously. However, it should be stressed that VS, SGPR, and PGPR are all guaranteed to produce collision-free solutions, while GLS is not.

Figure 4.2 displays the combined fraction of total reward collected for all agents as the number of agents is changed for a specific budget. The reward displayed is a fraction of the total available rewards in the graph for all vertices, therefore a value of 1 means that all vertices were visited and all rewards collected. As mentioned earlier, every agent has an equal amount of budget allocated to it, so the budget displayed is equivalent to $B \times |A|$. The results show that less agents with larger budgets are more effective than more agents with proportionally smaller budgets. This is expected because with less agents less coordination is necessary and each can be more impactful for its apportioned budget. In practice, however, this can be infeasible because it is

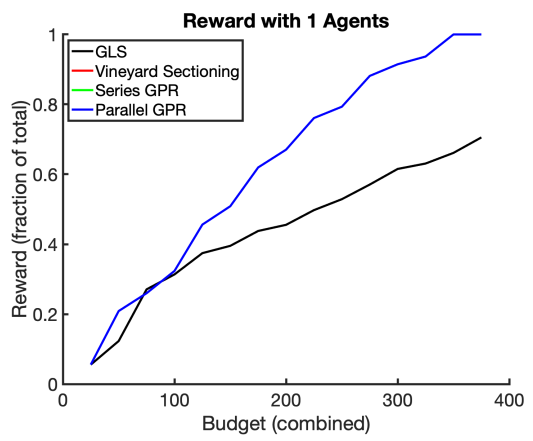
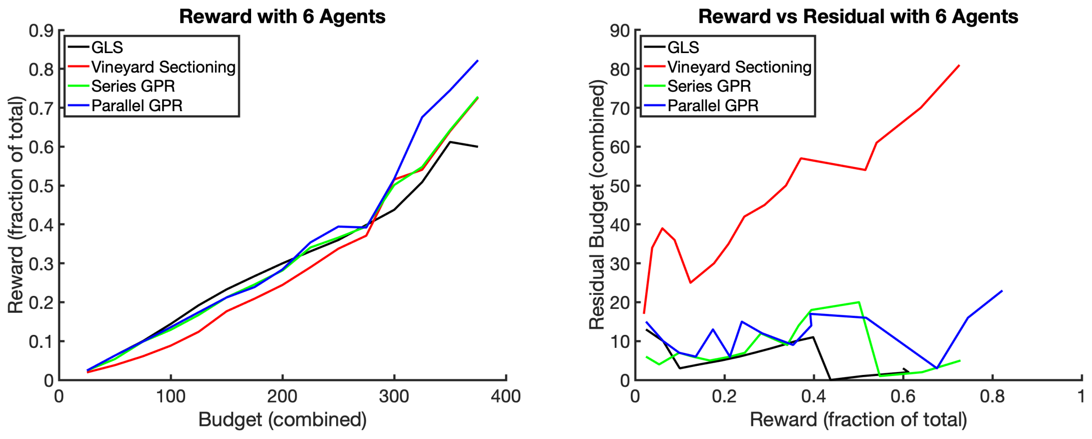


Figure 4.3: Budget vs fraction of total reward collected for 1 agent on a 12×25 graph. Note that the lines for Sectioning, Series, and Parallel are completely overlapping.

technically not viable to deploy fewer robots with the autonomy of more robots. The results of the various tests show that each of the AGTOP specific methods perform better than GLS in most cases, with the exception of diminutive budgets where GLS is an equal performer.

When the problem is reduced to the single agent case, Figure 4.3 shows that for each of the AGTOP algorithms, the collected reward is equivalent. This makes sense because all of these algorithms reduce to the original GPR algorithm Algorithm 3.2 when there is only one agent. All of the looping done to handle each agent is reduced to a single run, and there is no worry of possible collisions. In this case, it can be seen that VS, SGPR, and PGPR greatly outperform GLS for larger budgets.

Figure 4.4 shows the case where 6 agents are considered while the budget is varied. Figure 4.4a displays the overall reward for a given budget with each of the methods. At some budgets, each of the tested methods becomes the top performer in terms of reward collected, however generally PGPR is the dominant solver and VS is the inferior. Regardless, through most of the tested budgets excluding larger budgets, each of the algorithms collect a similar amount of reward. Examining only AGTOP methods, there is a maximum gap between VS and PGPR of 13.52% when $B = 325$. When GLS is considered, the largest gap is 47.25% between GLS and PGPR when $B = 375$. Figure 4.4b shows the amount of leftover (unused) budget after collecting a certain fraction of the reward. This is useful to highlight inefficiencies for each algorithm that are not apparent from looking at reward results alone. Residual budget emerges when agents need to end their paths at v_g and do not have enough budget to visit a new unvisited vertex. An optimal algorithm will have minimal residual as the fraction of total reward collected increases, with small spikes where graph structure prevents efficient use of budget, and a maximum reward at 1 where residual budget increases towards infinity. SGPR, PGPR, and GLS all have low residuals suggesting they coordinate agents efficiently, however GLS does not reach as high of a maximum reward as the other two. It is clear that VS does not coordinate agents adequately,



(a) (b)

Figure 4.4: Results on a 12 × 25 graph with 6 agents for various budgets. (a) The reward for each budget tested. (b) The residual budget left over at the end of each path.

as there is a lot of unused budget that is not spent gathering additional rewards. The results of reward collection per number of agents and budget, combined with the residual plot suggest that VS, SGPR, and PGPR are at least as economical as GLS, meanwhile the next set of tests shows they are not limited to graphs of this size and can scale up to more vertices.

4.4.2 VS, SGPR, and PGPR

The second set of tests compared the AGTOP algorithms to on the full-sized graph of $w = 240$ rows with $l = 500$ vertices per row, for a total of $120,000$ vertices. The same tests were performed, varying the number of agents and available budget. As before, none of the built paths had any overlapping use of edges so no collisions could occur between agents, since the methods were built to avoid this.

Figure 4.5 shows how the number of agents working on an AG impacts the amount of reward collected for a given budget. As before, more agents with proportionally smaller budgets generally collect less reward, because they cannot coordinate as efficiently as fewer agents. Similarly to the smaller test cases, SGPR and PGPR are almost equivalent for much of the tested number of agents, with PGPR performing slightly better, and VS is noticeably less productive.

Figure 4.6 exhibits the overall efficiency of the methods with a large quantity of agents. Figure 4.6a shows the reward of each method for a given budget with $|A| = 50$. For any input budget, SGPR and PGPR are nearly equivalent, with PGPR doing slightly better overall while VS lags behind in reward collection. With $B = 150,000$, PGPR accumulates 95.7% of the rewards, SGPR obtains 94.5%, and VS gathers 87.7% of the total available rewards. These results are similar with any number of agents (tested with 1 to 100 agents), however the gap between the methods converge

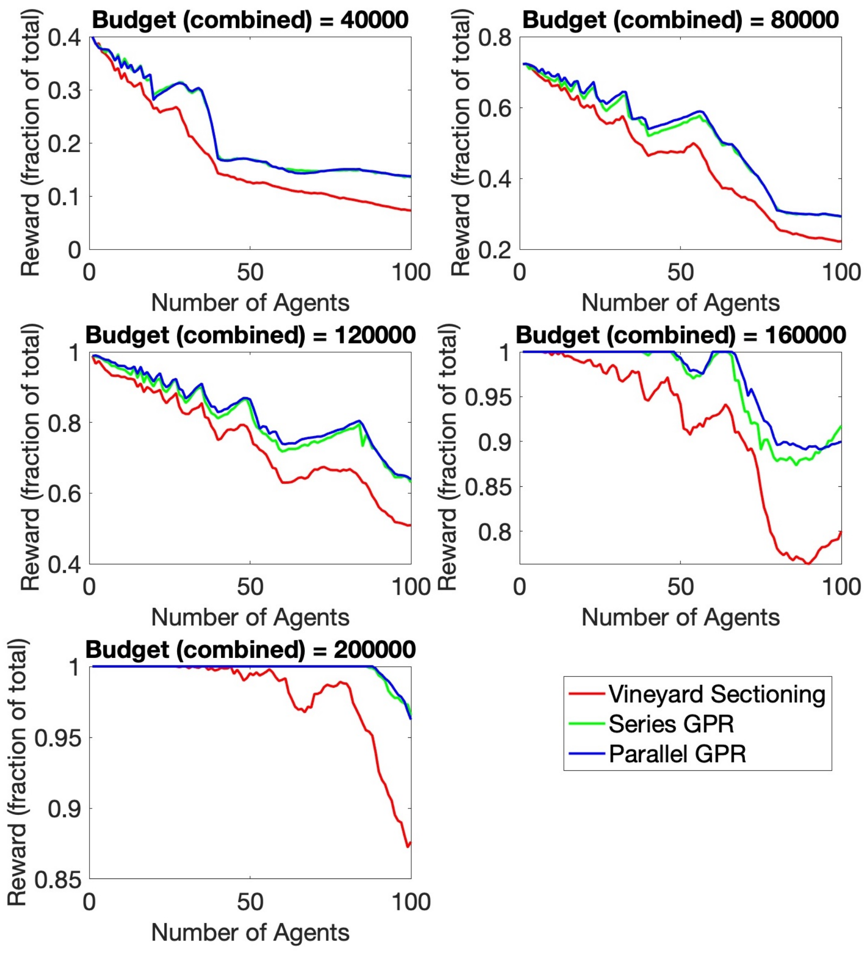


Figure 4.5: The fraction of available reward collected by a number of agents for a given fixed budget on a full-sized 240 × 500 graph. Note there are some overlapping lines where values are the same between multiple algorithms.

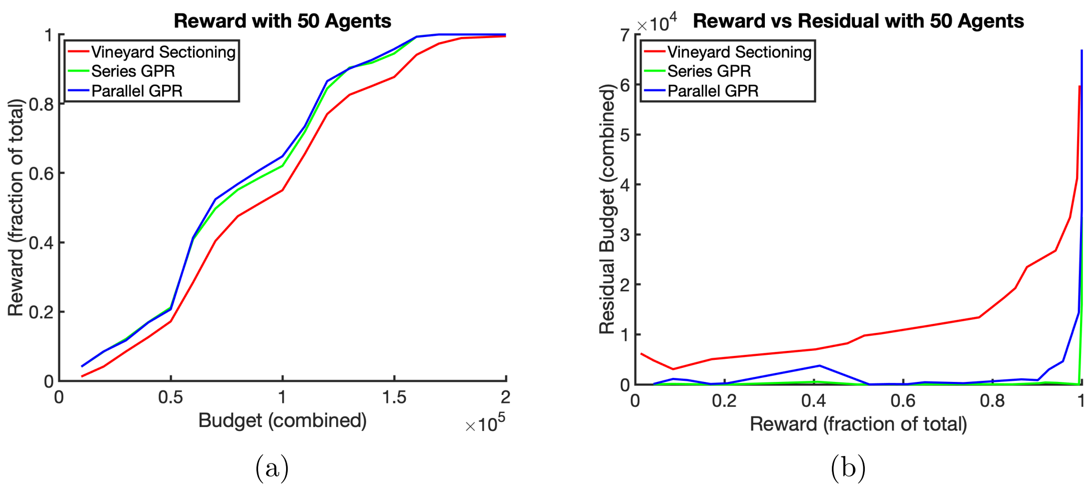


Figure 4.6: Results on a 240×500 graph with 50 agents for various budgets. (a) The reward for each budget tested. (b) The residual budget left over at the end of each path.

to nothing as $|A|$ approaches 1. The residual budget for the same tests are shown in Figure 4.6b and conveys an interesting result, showing that SGPR has the least unused budget throughout all tested budgets with PGPR having slightly more. This is because PGPR is able to coordinate each agent more effectively than SGPR and ends up in situations where no more vertices can be visited by any agent. Regarding VS, a large portion of its budget is wasted after a certain point. This is likely due to the compartmentalized usage of agents, where each agent can only visit a certain section of the graph. Because of the way the graph is sectioned off proportionally according to available rewards, some sections can be larger than others. When an agent in one of the smaller sections collects all the rewards available, it is stuck doing nothing and thus wastes the rest of its allocated budget, while other agents do not have enough budget to finish collecting available rewards in their sections. This proves that VS is rather wasteful and less efficacious than SGPR and PGPR, which are almost identical in performance on large AGs.

Finally, each of the three AGTOP methods were subjected to more tests on the same size AGs with different underlying vineyard moisture data for the reward functions. The results of these tests exhibit comparable results, with the relative effectiveness of each method remaining the same. Some tests were also done in [123] on AGs representing another vineyard in central California of size 275×214, with results averaged over 10 different moisture maps, and the outcomes are consistent with those described here.

4.5 Conclusion

In this chapter the TOP was studied on the special class of graphs where vertices are arranged in a set of rows or aisles. The problem is NP-hard on these graphs just like its single agent sibling the OP. Additionally, a robot coordination problem emerges when robots operation in environments structured like these graphs are not able to pass each other within rows without risk of collision. Three heuristic methods were developed specifically for team orienteering on these types of graphs - VS, SGPR, and PGPR - which build paths for each robot agent specifically evading collision. These were then compared to GLS, a general case heuristic method for the TOP that was not purposefully designed for AGs and does not have collision evasion. Tested using soil moisture data for rewards on a graph designed around a real-life vineyard, the three specially developed methods were proven to be more robust and better performers than GLS. These methods were capable of handling AGs containing hundreds of thousands of vertices and problems containing 100 agents, showing their practicality for the specific use case of precision agriculture.

Chapter 5

Bi-Objective Orienteering in Vineyards

Previous chapters looked at solution methods for single-objective orienteering on graphs with aisle-like row structure. A two objective (bi-objective) extension of the OP is examined in this chapter, with a focus on finding solutions on AGs. The work presented here was originally introduced in [122].

5.1 BOOP Background

Orienteering for two objectives, in a problem called the BOOP, is discussed in this section. Bi-objective orienteering is a sub-type of the MOOP, which itself is a type of generalized OP. Here, the BOOP is formalized for routing on general types of graphs and then the problem on AGs is briefly described.

There are two main sub-types of the BOOP, dual maximization and objective constraint, both of which share a number of common characteristics. As in the single-objective OP, the BOOP is given an undirected graph $G(V, E)$ with a cost function $c : E \to \mathbb{R}_{\geq 0}$ defined over its edges. Assuming a given budget B, a path $\mathcal{P}$ can be build over the graph that starts at some vertex v_s and ends at a goal location v_g whereby the total cost of the path is less than or equal to the budget $C(\mathcal{P}) \leq B$. However, unlike the OP, the BOOP has two reward functions defined for its vertices, $r_1 : V \to \mathbb{R}_{\geq 0}$ and $r_2 : V \to \mathbb{R}_{\geq 0}$. Like in the OP, both sets of rewards can be collected only once ($R_1(v)$ and $R_2(v)$ are both collected simultaneously when v is visited), and visiting a vertex multiple times does not grant more reward. These reward functions may be correlated, uncorrelated, anti-correlated, or some mixture of those. However they are defined, the BOOP asks to find a path visiting a subset of the vertices V which fulfills some goal over the two reward functions, the optimum of which is found somewhere on the Pareto frontier of solutions. The goal of the problem depends on the end user's preference and therefore can be different depending on various criteria.

5.1.1 Dual Maximization BOOP

The Dual Maximization BOOP (DMBOOP) seeks to maximize results for both objectives. The aim of the DMBOOP is to determine a path $\mathcal{P}$ of cost $C(\mathcal{P}) \leq B$ that maximally collects both sets of rewards for each vertex visited, i.e. $R_1(\mathcal{P})$ and

$R_2(\mathcal{P})$. If the second reward function is set to 0 for all vertices, then the problem becomes the OP, proving the the DMBOOP is a multi-objective extention of the OP and is therefore NP-hard. For the case of AGs, the problem does not change, and thus the AG version can be defined as follows:

> **Aisle Graph Dual Maximization Bi-Objective Orienteering Problem (AGDMBOOP):** Given a graph $G(V, E) = AG(w, l)$ with cost function C and two reward functions R_1, R_2, vertices within the graph $v_s, v_g \in V$, and a constant B, find a path $\mathcal{P} \subset V$ with cost no more than B that begins at v_s and ends at v_g which independently maximizes the cumulative of both reward functions $R_1(\mathcal{P})$ and $R_2(\mathcal{P})$ for visited vertices.

In general, such a requirement to maximize both reward functions is impossible since it is often the case that the reward functions are independent and uncorrelated. When dealing with dual maximization problems, or any multi-objective optimization problem that has no clear predilection towards one objective in particular, there are a few common approaches to resolving solutions. Creation of a Pareto frontier (see [93]) is the most comprehensive method, which involves finding a set of solutions with slightly different outcomes which allow the end user to choose an appropriate solution. In orienteering this means finding multiple paths through the graph, a process that may be extremely time consuming given that even finding one optimal path for a single objective is NP-hard. Other methods seek to find efficient solutions by characterizing the relationship between the reward functions such that decision variables are chosen based on how positively they impact both reward functions.

One frequently used scheme is blending both objective functions into a single weighted function, such as with linear combination. This typically works in practice, however it is difficult to qualify how well it works without first solving the problem across multiple weighting parameters effectively creating a pseudo-Pareto frontier. To solve the BOOP using weighted linear combination, one can update the reward for each vertex to reflect a new reward function:

$$R(v) = \alpha R_1(v) + (1 - \alpha) R_2(v) \tag{5.1}$$

where α is a weighting parameter between 0 an 1. This can be especially useful because it is flexible enough to use with any orienteering solution method, including those exhibited in Chapter 3 and Chapter 4. However, linear combination has one major drawback associated with it; the sum of reward functions is physically meaningless. This is because the reward functions may represent different real world concepts. If they represented the same thing, then there would be no need for separate reward functions. This makes it challenging to chose the right value of α for an adequate mixing, hence why evaluating for multiple weight values is necessary.

5.1.2 Objective Constraint BOOP

The Objective Constraint BOOP (OCBOOP) seeks to meet a lower bound on one objective while maximizing the other. The aim of the OCBOOP is to determine a path $\mathcal{P}$ of cost $C(\mathcal{P}) \leq B$ that maximally collects rewards for the first reward function $R_1(\mathcal{P})$ while meeting or exceeding a lower bound on a the second reward function $R_2(\mathcal{P}) \geq r_{min}$. This problem is also NP-hard, given that it is simply an extension of the OP with an additional constraint, which can be made back into the OP by making $r_{min} = 0$. There are no special considerations for solving this problem on AGs, and thus the AG version can be defined as follows:

> **Aisle Graph Objective Constraint Bi-Objective Orienteering Problem (AGOCBOOP):** Given a graph $G(V, E) = AG(w, l)$ with cost function C and two reward functions R_1, R_2, vertices within the graph $v_s, v_g \in V$, and a constant B, find a path $\mathcal{P} \subset V$ with cost no more than B that begins at v_s and ends at v_g which maximizes cumulatively the first reward function $R_1(\mathcal{P})$ and satisfies a lower bound on the cumulative sum of the second reward function $R_2(\mathcal{P}) \geq r_{min}$ for visited vertices.

Unlike with dual maximization orienteering, solving the OCBOOP does not require a subjective method for evaluating or combining the reward functions. This is because only one reward function is optimized; the other is merely constrained to a certain minimum value. This makes the objective constraint version a preferential type of BOOP to an end user, since the reward function has a distinct physical interpretation. To this end, an integer linear program can easily be written to find an optimal solution to the OCBOOP, by adding in the additional constraint on R_2 to the formulation given in section 3.1, however heuristic methods are less easily modified for this purpose.

5.2 Extending GPR to the BOOP

When it comes to solving BOOPs on problems modeled after precision agriculture in vineyards, the large size of the resulting AGs requires the use of heuristics. Therefore the GPR heuristic given in section 3.3 was again extended, this time to solve either the AGDMBOOP or the AGOCBOOP. These methods were first presented in [122].

5.2.1 Dual Maximization Methods

The methods presented in this subsection solve the AGDMBOOP. They are meta-methods, which can incorporate any method used to solve AGOPs including generalize heuristics such as the S-Algorithm, however for problems of large sizes, only GPR is practical to use. They work by attempting to balance the two objectives through assigning a weighting value to each objective, which is used to allocate resources according to some “importance” given by the user. For each method given, this weighing parameter is α, which can take any value from 0 to 1. This is the only tuning parameter used for this class of algorithms; all other inputs are intrinsic to the problem.

Heuristic/Heuristic

A very simple way to utilize a heuristic algorithm built for single-objective orienteering to solve bi-objective problems is to run the heuristic twice, allocating a proportion of the budget to collecting R_1 rewards and using the rest of the budget to collect R_2 rewards. The Heuristic/Heuristic method solves the BOOP by doing exactly this, taking a parameter α used to determine the fraction the overall budget to allocate to the collection of each reward, i.e. $B_1 = \alpha B$ and $B_2 = (1-\alpha)B$. By running GPR in succession to collect each set of rewards, a single path can be performed by combining the two paths sequentially. The Heuristic/Heuristic method is displayed in Algorithm 5.7.

Algorithm 5.7 Dual Maximization Heuristic/Heuristic

```
Input: $AG(w,l)$, $R_1$, $R_2$, $C$, $B$, $v_s$, $v_g$, $\alpha$
Output: $\mathcal{P}$
1: $\mathcal{P}_1 \leftarrow$ GPR($AG(w,l)$, $R_1$, $C$, $\alpha B$, $v_s$, $v_g$)
2: Remove end of $\mathcal{P}_1$ until last traversed full-row or partial-row
3: for all $v_i \in \mathcal{P}_1$ do
4:     $R_1(v_i) \leftarrow 0$
5:     $R_2(v_i) \leftarrow 0$
6: $\mathcal{P}_2 \leftarrow$ GPR($AG(w,l)$, $R_2$ $C$, $(1-\alpha)B$, $\mathcal{P}_1(end)$, $v_g$)
7: $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$
8: return $\mathcal{P}$
```

First, GPR is run to collect rewards on R_1 using the budget allocated αB (line 1) to build $\mathcal{P}_1$. In this first run, both sets of rewards are collected simultaneously as $\mathcal{P}_1$ passes each vertex, however GPR builds $\mathcal{P}_1$ agnostic of R_2. Next, the end of $\mathcal{P}_1$ is removed, such that it stops at the vertex on the left or right side column where the final full-row or partial-row chosen to be included in the path exits (line 2). Then, for all visited vertices in $\mathcal{P}_1$, both reward functions are given values of 0 (lines 3-l5). This is to ensure that the second run of GPR does not attempt to revisit any vertices and collect their rewards again. Afterward, GPR is run a second time, specifically to collect R_2 rewards while being agnostic of but still collecting R_1 rewards using the leftover budget $(1-\alpha)B$ (line 6). Here, $\mathcal{P}_2$ starts at the last vertex in the first path built $\mathcal{P}_1(end)$. This allows a seamless transition so that the two paths can be joined sequentially into one path $\mathcal{P}$.

Algorithm 5.7 is comprised of two executions of the GPR heuristic, and therefore inherits is computational complexity, $\mathcal{O}(w^2 l)$. If any other AGOP solver were used, then it would inherit the complexity of that algorithm instead.

Heuristic/Knapsack

The next method, Heuristic/Knapsack, works by building and initial tour and augmenting it for extra rewards. As before, the parameter α allocates a proportion of the overall budget B to creating an initial tour with the GPR heuristic, focusing on gathering R_1 rewards, then the leftover budget $(1-\alpha)B$ is used to extend portions of the path or add new partial rows to it for gathering R_2 rewards. The 01-knapsack algorithm is used to choose these extensions and new partial rows. Pseudo-code for the Heuristic/Knapsack method is outlined in Algorithm 5.8.

Algorithm 5.8 Heuristic/Knapsack

```
Input: $AG(w,l)$, $R_1$, $R_2$, $C$, $B$, $v_s$, $v_g$, $\alpha$
Output: $\mathcal{P}$
 1: $\mathcal{P} \leftarrow$ GPR($AG(w,l)$, $R_1$, $C$, $\alpha B$, $v_s$, $v_g$)
 2: for all $v_i \in \mathcal{P}$ do
 3:     $R_2(v_i) \leftarrow 0$
 4: $\mathcal{KL} \leftarrow \emptyset$
 5: for all $v_i \in V$ where $R_2(v_i) > 0$ do
 6:     Find minimal cost insertion of $v_i$ to $\mathcal{P}$ and add to $\mathcal{KL}$
 7: while True do
 8:     $\mathcal{CL} \leftarrow$01-knapsack($B - C(\mathcal{P})$, $\mathcal{KL}$, $R_2$)
 9:     for $i \leftarrow 1 \ldots |\mathcal{CL}|$ do
10:         for $j \leftarrow i+1 \ldots |\mathcal{CL}|$ do
11:             if $\mathcal{CL}(i)$,$\mathcal{CL}(j)$ overlap and $|\mathcal{CL}(i)| > |\mathcal{CL}(j)|$ then
12:                 $R_2(i) \leftarrow R_2(i) + R_2(j)$, $R_2(j) \leftarrow 0$
13:                 Remove $\mathcal{CL}(j)$ from $\mathcal{KL}$
14:             else if $\mathcal{CL}(i)$,$\mathcal{CL}(j)$ overlap and $|\mathcal{CL}(i)| < |\mathcal{CL}(j)|$ then
15:                 $R_2(j) \leftarrow R_2(j) + R_2(i)$, $R_2(i) \leftarrow 0$
16:                 Remove $\mathcal{CL}(i)$ from $\mathcal{KL}$
17:                 Break
18:     if $\mathcal{KL}$ did not change then
19:         Break
20: for $i \leftarrow 1 \ldots |\mathcal{CL}|$ do
21:     Insert $\mathcal{CL}(i)$ into $\mathcal{P}$
```

Initially, a path $\mathcal{P}$ is built over the AG using GPR seeking to maximize R_1 rewards with a proportion α of the total budget (line 1). All vertices which $\mathcal{P}$ visits then have their R_2 rewards set to 0, as these locations do not need to be visited again (lines 2-3). Next, for every vertex with R_2 rewards on the graph, the minimal cost insertion to the path is computed and stored in a list $\mathcal{KL}$ (lines 4-6). These insertions are detours to new vertices and back which make $\mathcal{P}$ longer but do not modify any other aspect of the path. Next, the main loop begins which chooses a subset $\mathcal{CL}$ of $\mathcal{KL}$ that maximizes the collected R_2 rewards for the leftover budget $B - C(\mathcal{P})$ (lines 7-19). The chosen detours are then inserted into $\mathcal{P}$ giving the path to be returned. Note that if some budget remains then GPR can be run again as in Algorithm 5.7 to utilize the rest for finding R_1 or R_2 rewards by extending the path to some more vertices.

The main loop, where detours are chosen, deserves more explanation. The set of possible detours $\mathcal{KL}$, along with their insertion costs and R_2 rewards, are synthesized into a 01-knapsack problem, which chooses the most profitable subset of detours $\mathcal{CL}$ for the remaining budget $B - C(\mathcal{P})$ (line 8). Any chosen detours that are overlapping occur in the same row, and thus can be combined into a single detour (lines 11-17). This combination simply means removing the shorter overlapping detours from consideration and adding their rewards to the longer one, as the longer one necessarily visits the same vertices and thus actually collects those rewards as well. If overlaps are found, the 01-knapsack solver is rerun to find a more efficient solution, whereas if none are found the main loop terminates (lines 18-19).

The computational complexity of this algorithm is $\mathcal{O}(w^2l+\alpha B\cdot wl+(1-\alpha)B\cdot w^2l^2)$. The first term comes from using GPR to create an initial path, the second is from finding minimal cost insertions to the path (max length of $\mathcal{P}$ times the number of possible locations for R_2), and the last is from repeatedly using dynamic programming to solve the 01-knapsack problem up to wl times. Considering the possibility of $\alpha = 0$, the worst case complexity reduces to $\mathcal{O}(B \cdot w^2l^2)$. Finally, it should be noted that because collection of R_2 rewards is done by inserting detours to the initial path, it makes little sense to have $\alpha \leq 0.5$ because the detours will overwhelm the initial path. Therefore, reward functions that are more preferential to maximize should be set as R_1 for this algorithm and $\alpha \leq 0.5$ should not be used.

Weighted Addition

The next approach to maximize two reward functions at once on an AG is to combine them into a single reward function over which GPR is run. This method, called Weighted Addition, assigns a weight to each reward function that allows an end user to manipulate the balance of collecting one reward versus the other with a single parameter α. To keep the new reward function from being biased towards one of the two original functions (excluding the weight parameter), the rewards are normalized for each vertex. The new reward function used as input into GPR is given as:

$$R(v) = \alpha \cdot \frac{R_1(v)}{\sum_{v_i \in V} R_1(v_i)} + (1-\alpha) \cdot \frac{R_2(v)}{\sum_{v_i \in V} R_2(v_i)} \quad \forall v \in V \tag{5.2}$$

This new reward function is similar to the expected information gain function in [32], where two separate functions are normalized and combined as a weighted sum. Due to the lack of a concrete physical interpretation to underpin a particular desired quantitative outcome, the weight parameter α must be carefully selected. As the new combined reward function is used directly by GPR when building a path, the worst case computational complexity for using the weighted addition is the same as using GPR, $\mathcal{O}(w^2l)$.

5.2.2 Objective Constraint Methods

The methods presented in this subsection solve the AGOCBOOP. Like the AGDM-BOOP methods, they are also meta-methods that can make use of AGOP solvers other than GPR, but here are presented as using GPR for practicality reasons. Since objective constraint orienteering attempts to build paths that meet a minimum requirement for the R_2 reward function, the only required user definable parameter is r_{min}.

Heuristic/Heuristic

Like the AGDMBOOP, a very simple way to solving the AGOCBOOP is to utilize a heuristic algorithm to build a path twice in a row, first until it satisfies the constraint and then again until the rest of the budget is consumed. This prioritizes collecting R_2 rewards until r_{min} is reached, and then focus is switched to collecting R_1 rewards. Because each objective function is assumed to be independent, and all possible rewards are positive in AGOP problems, achieving $R_2(\mathcal{P}) \geq r_{min}$ first means that the constraint will continue to be satisfied as long as the first part of $\mathcal{P}$ does not change. This method works similarly to that described in section 5.2.1, except the first run of GPR terminates immediately upon fulfilling the minimum bound constraint. Algorithm 5.9 shows how this method is implemented.

This method is essentially the same as Algorithm 3.2, with the only changes revolving around the two different reward functions. The cumulative sums from either side of the graph must be computed for each (lines 2-7), and they need to be updated after every iteration (line 39). The status of the constraint is checked before calculating the next full-row or partial-row to add to the path so that the appropriate reward function is used (lines 13-18, 23-28). Overall, the complexity of this algorithm is the same as GPR, i.e. $\mathcal{O}(w^2 l)$.

Bisection GPR

Running a heuristic twice in a row to collect two different sets of rewards works, however if one reward only needs to reach a minimum constrained value, then the possibility exists that the other reward is not optimally maximized. With this realization, it may not be necessary to satisfy the constraint immediately and it can instead be reached over the course of maximizing the other reward. It may be possible to satisfy the lower bound while devoting only a small proportion of resources to it, however the amount needed is difficult to anticipate. One method of finding the right balance of resources is to use a bisection search, running on any of the methods presented in subsection 5.2.1 and dynamically adjusting the parameter α. This approach is called Bisection GPR and is presented in Algorithm 5.10.

The Bisection GPR method can be used with any of the methods that solve the DMBOOP problem, Heuristic/Heuristic, Heuristic/Knapsack, or Weighted Addition. These methods all take a parameter α which is used to adjust the relative importance of the two reward functions, such that changing α will result in collecting more of one reward, depending on which direction it is changed in. This is the principle that Bisection GPR works on, calling the chosen DMBOOP method multiple times in a loop until a stopping criteria is met (lines 3-4). In each iteration, α is updated to be larger if r_{min} is not satisfied (lines 5-11), or smaller if r_{min} is satisfied (lines 12-14).

Algorithm 5.9 Objective Constraint Heuristic/Heuristic

```
Input: $AG(w, l)$, $R_1$, $R_2$, $C$, $B$, $v_s$, $v_g$
Output: $\mathcal{P}$
 1: $\mathcal{P} = \{v_s\}$
 2: for all $v_{i,j} \in V$ do
 3:     $R_{1,i,j} \leftarrow \sum_{n=j}^{l} R_1(i,n)$
 4:     $L_{1,i,j} \leftarrow \sum_{n=1}^{j} R_1(i,n)$
 5:     $R_{2,i,j} \leftarrow \sum_{n=j}^{l} R_2(i,n)$
 6:     $L_{2,i,j} \leftarrow \sum_{n=1}^{j} R_2(i,n)$
 7:     $feasible_{i,j} \leftarrow$ True
 8: while $any(feasible) =$ True do
 9:     for all $v_{i,j} \in V$ do
10:         if $feasible(i,j,\mathcal{P}) \neq$ True then
11:             $feasible_{i,j} \leftarrow$ False
12:     for all $feasible_{i,j}$ do
13:         if $R_2(\mathcal{P}) < r_{min}$ then
14:             $R'_{i,j} \leftarrow R_{2,i,j}/C(\mathcal{P}(end), v_{i,j}, v_{i,n})$
15:             $L'_{i,j} \leftarrow L_{2,i,j}/C(\mathcal{P}(end), v_{i,j}, v_{i,1})$
16:         else
17:             $R'_{i,j} \leftarrow R_{1,i,j}/C(\mathcal{P}(end), v_{i,j}, v_{i,n})$
18:             $L'_{i,j} \leftarrow L_{1,i,j}/C(\mathcal{P}(end), v_{i,j}, v_{i,1})$
19:     for $i \leftarrow 1$ to $w$ do
20:         if $feasible(i,\mathcal{P}) \neq$ True then
21:             $feasible_i \leftarrow$ False
22:     for all $feasible_i$ do
23:         if $R_2(\mathcal{P}) < r_{min}$ then
24:             $R'_{i,1} \leftarrow R_{2,i,1}/C(\mathcal{P}(end), v_{i,1})$
25:             $L'_{i,n} \leftarrow R_{2,i,n}/C(\mathcal{P}(end), v_{i,n})$
26:         else
27:             $R'_{i,1} \leftarrow R_{1,i,1}/C(\mathcal{P}(end), v_{i,1})$
28:             $L'_{i,n} \leftarrow R_{1,i,n}/C(\mathcal{P}(end), v_{i,n})$
29:     if $\mathcal{P}(end)_j = l$ then
30:         $best \leftarrow \arg\max R'_{i,1}, R'_{i,j}$
31:         $side = l$
32:     else
33:         $best \leftarrow \arg\max L'_{i,l}, L'_{i,j}$
34:         $side = 1$
35:     to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $best$
36:     if $\mathcal{P}(end)_j \neq 1$ or $l$ then
37:         to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $v_{i,side}$
38:     $feasible_{best} \leftarrow$ False
39:     update $R(1,i,j)$, $L(1,i,j)$, $R(2,i,j)$ and $L(2,i,j)$ for $i = best_i$
40: to $\mathcal{P}$ append path from $\mathcal{P}(end)$ to $v_g$
41: return $\mathcal{P}$
```

Algorithm 5.10 Bisection GPR

```
Input: $AG(w,l)$, $R_1$, $R_2$, $C$, $B$, $v_s$, $v_g$
Output: $\mathcal{P}$
1: $\alpha_h = 1;\ \alpha_l = 0;\ \alpha = 0.5$
2: $\mathcal{P} = \emptyset$
3: while $\alpha_h - \alpha_l > \varepsilon$ do
4:     $\mathcal{P}_{next} \leftarrow$ AGDMBOOP-GPR($AG(w,l)$, $R_1$, $R_2$, $C$, $B$, $v_s$, $v_g$, $\alpha$)
5:     if $R_2(\mathcal{P}_{next}) \geq r_{min}$ and $R_1(\mathcal{P}_{next}) > R_1(\mathcal{P})$ then
6:         $\mathcal{P} = \mathcal{P}_{next}$
7:         $\alpha_l = \alpha$
8:         $\alpha = (\alpha + \alpha_h)/2$
9:     else if $R_2 \geq r_{min}$ then
10:        $\alpha_l = \alpha$
11:        $\alpha = (\alpha + \alpha_h)/2$
12:    else
13:        $\alpha_h = \alpha$
14:        $\alpha = (\alpha + \alpha_l)/2$
15: return  return $\mathcal{P}$
```

The stopping criteria for the main loop is shown here as ε, the minimal allowed change in α, however it could also be a minimal change in one of the reward values or receiving the same $\mathcal{P}$ two iterations in a row. Eventually, this causes Algorithm 5.10 to terminate, which gives a computational complexity of $\mathcal{O}(X \cdot \log(1/\varepsilon))$, where X is the complexity of the bi-objective GPR method used within.

5.3 Experimental Evaluation

As with the AGOP and AGTOP, assessments were made of the methods presented in section 5.2 using a real-life vineyard on which robotics problems could be simulated. For the bi-objective problems, two reward functions had to be defined, with the first one representing the irrigation rewards given for a robot adjusting irrigation emitters at each vine like the experiments in section 4.4 and section 5.3. The second reward function was made to represent the benefit of collecting additional soil moisture samples using a robot with an onboard probe, much like the one shown in Figure 5.1. Results shown in this section were originally showcased in [122].

5.3.1 Vineyard AG with Two Rewards

The algorithms given in section 5.2 were assessed using a number of different simulations built using data collected from a commercial vineyard located near Madera, California. The vineyard on which the tests were performed has $w = 275$ rows and $l = 214$ vines per row for a total of $58,850$ vines. Rows were spaced 8 feet (2.43 meters) apart and vines were spaced 6 feet (1.83 meters) apart, giving a plot size of roughly

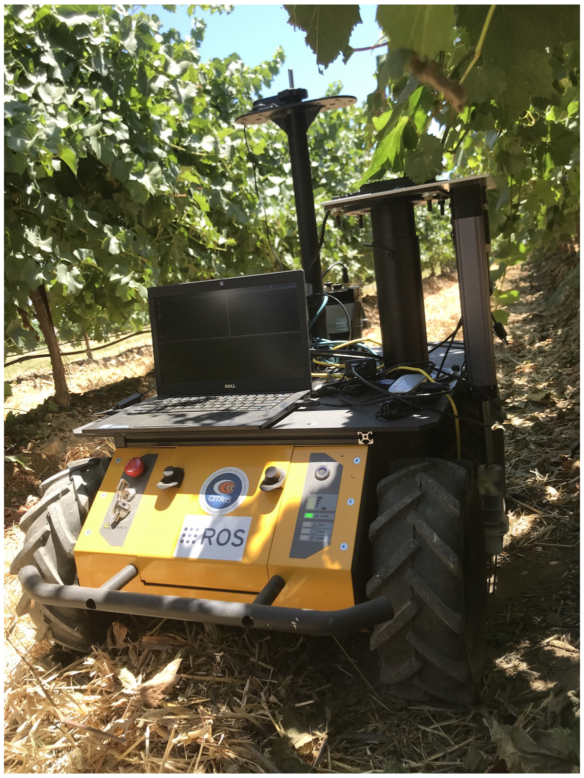


Figure 5.1: A prototype robot built with the Clearpath Husky platform and equipped with a linear actuator mounted soil moisture probe connected to a data logger.

64 acres (25.9 hectare). Soil moisture values were sampled manually across this vineyard at 72 uniformly spaced locations. The instrument used was a Hydrosense HS2P manufactured by Campbell Scientific, which is capable of GPS logging and taking multiple measurements at every location. In total 9 different data sets were collected, and for each sample locations were probed within a 3 hour window during a single summer day to produce an accurate snapshot of soil conditions. Figure 5.2 shows an aerial view of sample locations of the vineyard.

Kriging, a Gaussian Process interpolation procedure (see [94]), was used to obtain a continuous map of soil moisture values as well as estimation variance values for every vine location in the vineyard. These were used to create the reward functions necessary for the AGBOOP problems. The absolute value of the desired soil moisture value M minus the measured interpolated values $m(v)$ gave one set of rewards, $R_1(v) = |M - m(v)|$, and the variance values were directly used for the other set of rewards, $R_2(v) = \sigma^2(v)$. In this case, R_1 represented the irrigation rewards for adjusting irrigation emitters at each vertex, and R_2 represented the sampling rewards for taking an additional soil moisture sample. An interesting property of variance calculated by Kriging of uniformly spaced samples is that the variance values are also uniformly spaced. Because of this, some vertices were randomly given $R_2(v) = 0$ and noise was added to the remaining vertices to discourage uniformity. An example of heat maps displaying both types of rewards is given in Figure 5.3.

A range of 13 different budgets B were tested along with various values for α and r_{min} to thoroughly test each algorithm on each data set. The results shown are averages for all data sets across a single value of B and α or r_{min}. This was done to eliminate fluctuations due to reward disparities and clearly exhibit the results without bias for any single simulation or data set.

5.3.2 Dual Maximization

The three methods outlined for solving the AGDMBOOP - Heuristic/Heuristic, Heuristic/Knapsack, and Weighted Addition - were run on the test AG with different data sets while varying the total budget B and user parameter α. The results of these tests can be seen in Figure 5.4 and Figure 5.5. All of the reward values are normalized such that a value of 1 on the y-axis coincides with collecting 100% of the rewards for the corresponding reward function. For comparison, GPR from section 3.3 is shown as well, which was always provided with the full budget and does not change depending on α. This was done to provide results of collecting both rewards using a solver that only considers R_1 irrigation rewards when building an output path and is agnostic of R_2 sampling rewards. $\alpha \leq 0.5$ was not tested because it is functionally similar to switching R_1 and R_2 rewards and testing with $1 \geq \alpha \geq 0.5$.

With the variation of α, a trend emerges in the irrigation reward R_1 results of each method, shown by Figure 5.4. For values of α close to 1, all methods collect nearly identical irrigation rewards and follows closely the results of the standard GPR algorithm. This is because each each method is based on GPR, and at $\alpha = 1$ they are

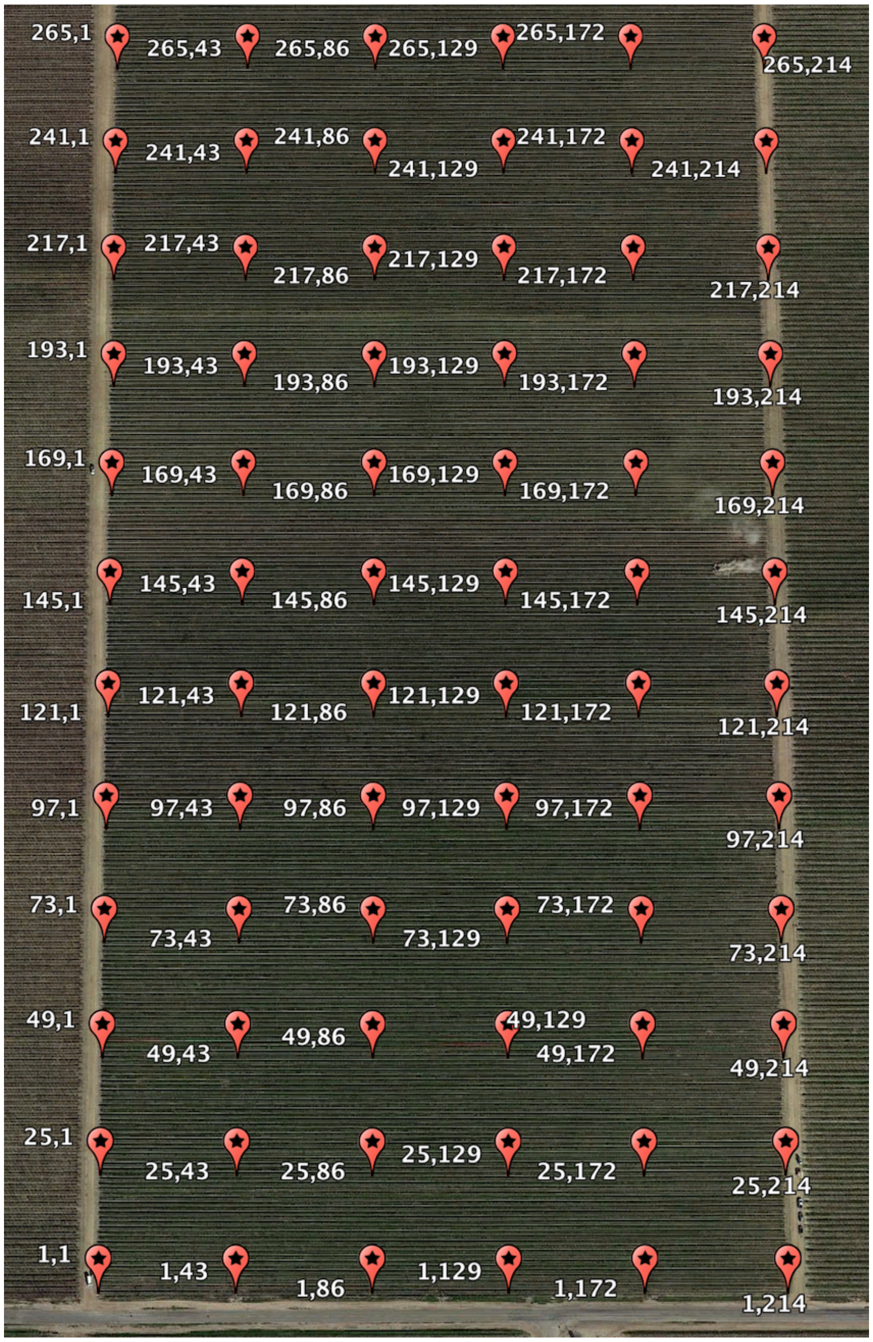
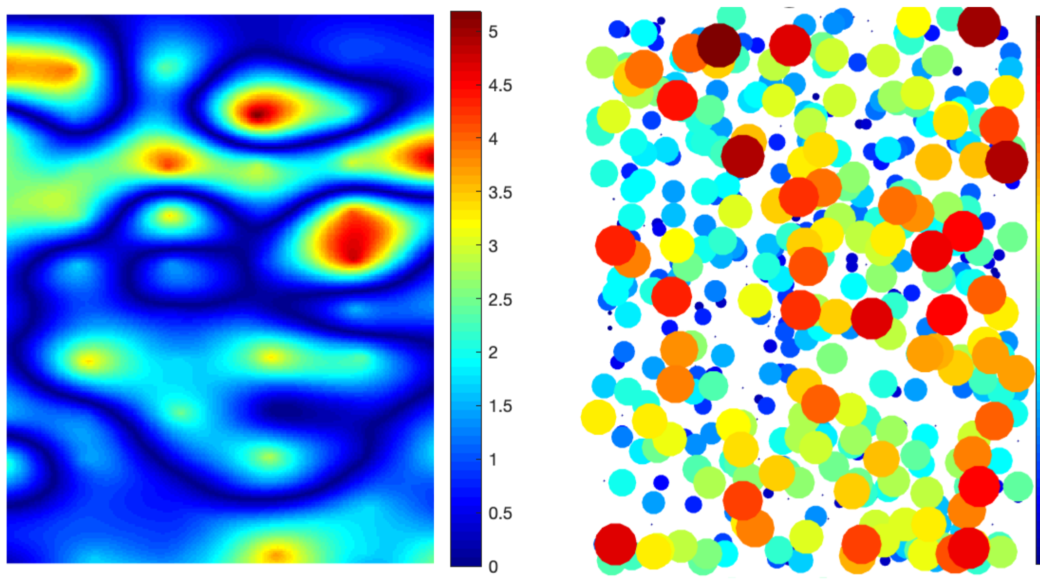


Figure 5.2: Locations where soil moisture data was collected in a vineyard outside of Madera, California.



(a) (b)

Figure 5.3: (a) Irrigation rewards R_1 and (b) Sampling rewards R_2 for every vertex in the graph. Note that irrigation rewards are available at every vertex while sampling rewards are available at a limited number of vertices. Large dots are used for enhanced visual depiction of the locations, and white space or lack of a dot signifies a location where $R_2(v) = 0$.

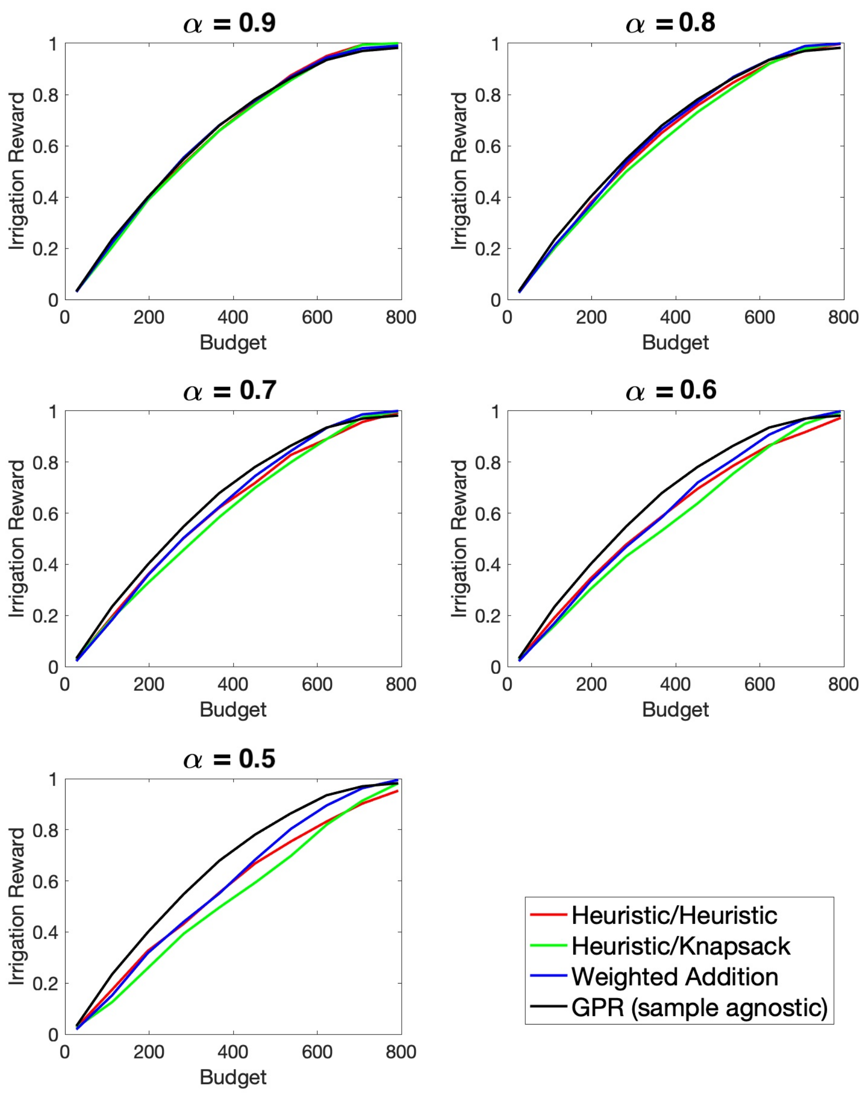


Figure 5.4: Irrigation rewards R_1 collected using each of the proposed AGDMBOOP methods with different values of parameter α.

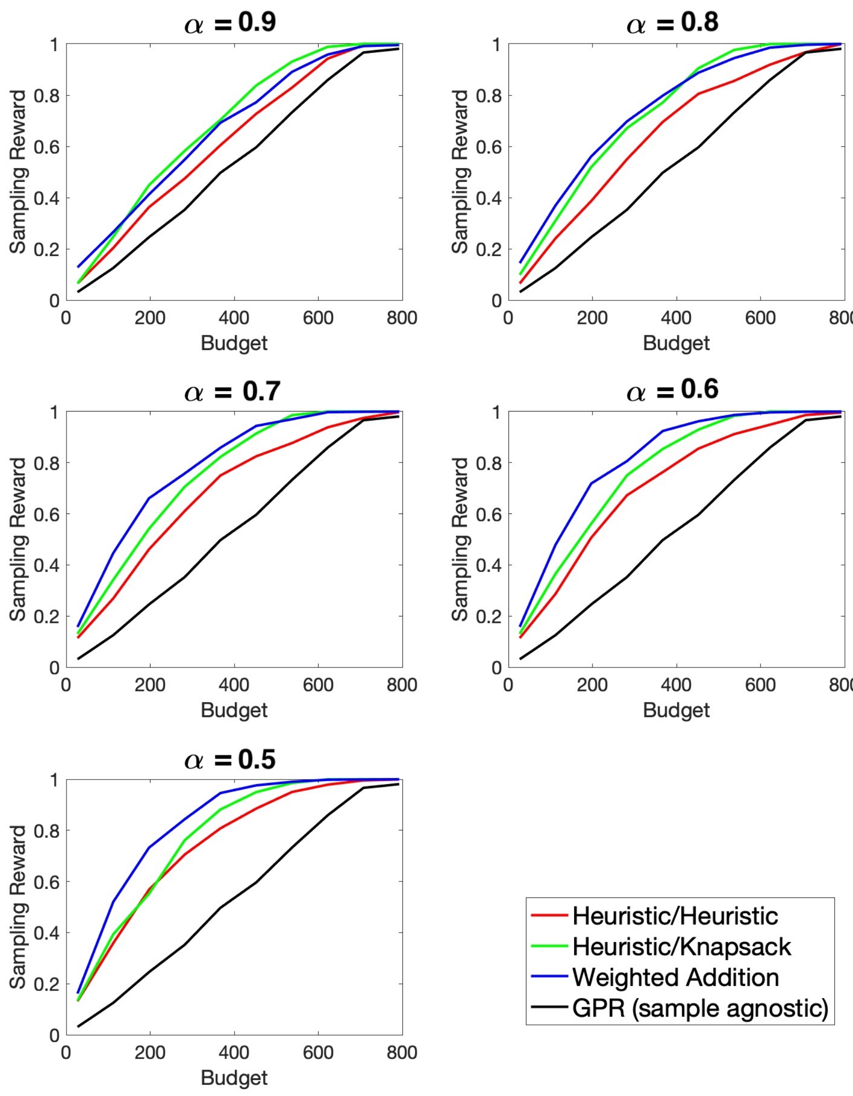


Figure 5.5: Sampling rewards R_2 collected using each of the proposed AGDMBOOP methods with different values of parameter α.

reduced to exactly the same algorithm. Intuitively, the opposite is true, where the closer to $\alpha = 0$ the methods diverge from each other. Particularly, the differences are seen easily when $\alpha = 0.5$, where GPR collects the most irrigation rewards through all budgets tested, Heuristic/Knapsack collects the least through the majority of tested budget, and Weighted Addition is tied with Heuristic/Heuristic initially but eventually wins.

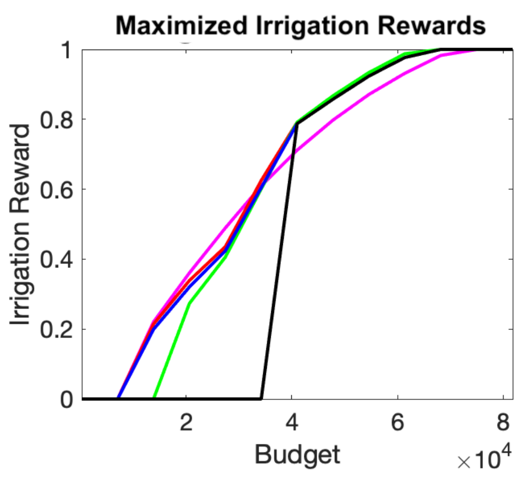
There is a much different trend in the results of each method regarding changes of α effecting the sampling reward R_2 results, shown by Figure 5.5. There is a clear disparity between all methods at every value of α tested, with a large separation between GPR which is agnostic to sample rewards and the other algorithms which are not. Even when $\alpha = 0.9$, the difference is significant and much more sampling rewards are collected, whereas smaller values of α just increase the gap. When $\alpha \geq 0.8$, the best method at collecting sampling rewards seems to be Heuristic/Knapsack but only at larger budgets. When α decreases, Weighted Addition eventually becomes the most effective for the entire range of tested budgets. The Heuristic/Heuristic method always collects more than GPR, but almost always less than every other method.

Evidently, when $\alpha \geq 0.9$, representative of placing a large importance on collecting R_1 rewards, the Heuristic/Knapsack method is best because it collects a substantial amount of R_2 rewards while also nearly matching GPR's performance on R_1 rewards. Moreover, as α gets smaller and R_2 rewards become more important, Weighted Addition becomes the preferred choice.

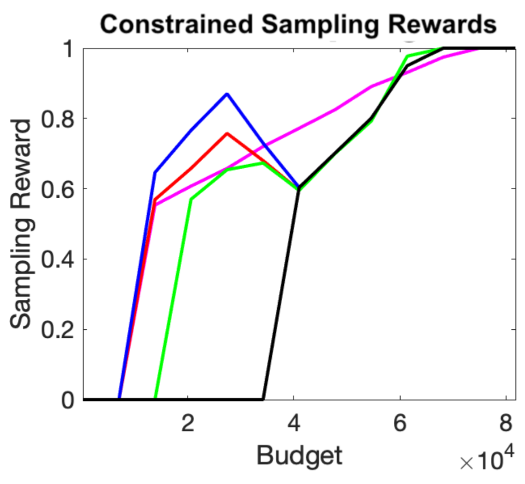
5.3.3 Objective Constraint

The AGOCBOOP methods - Heuristic/Heuristic and Bisection GPR using each of the AGDMBOOP algorithms - were run on the test AG with different datasets while varying the total budget B. Performance was considered while constraining r_{min} to be 50% of all R_2 rewards, and therefore any output path collecting less than that was considered a failure thus receiving 0 rewards for both R_1 and R_2. Figure 5.6 shows tests where the sampling rewards were set as R_2 and constrained, while Figure 5.7 shows tests where irrigation rewards were set as R_2 and constrained.

In the case of maximizing irrigation rewards while constraining sampling rewards, the performance of each proposed method is similar. GPR, which does not attempt to collect sampling rewards and does so passively, fails for half of the budgets to meet r_{min}, and therefore is shown in Figure 5.6 as having collected zero rewards. The Bisection method using Heuristic/Knapsack (green) did not satisfy the minimum sampling reward requirement on the two smallest budgets, while the rest of the AGOCBOOP did not satisfy the constraint on only the smallest budget. With larger budgets, the Bisection method using either Heuristic/Heuristic (red), Heuristic/Knapsack (green), or Weighted Addition (blue) had similar performance, however with lower-range budgets the Heuristic/Knapsack Bisection search performed worse than the others. Bisection using Heuristic/Heuristic seems to be the top overall performer because it met the minimum bound in all tests while collecting the most or



(a)



(b)

Figure 5.6: (a) Irrigation rewards R_1 and (b) sampling rewards R_2 for the Heuristic/Heuristic and Bisection AGOCBOOP methods, where $r_{min} = 0.5 \cdot \sum_{v \in V} R_2(v)$. **Legend**: Magenta is Heuristic/Heuristic, red is Heuristic/Heuristic Bisection, green is Heuristic/Knapsack Bisection, blue is Weighted Addition, and black is GPR.

nearly the most irrigation rewards.

In the case of maximizing sampling rewards while constraining irrigation rewards, the results differ appreciably. Each of the methods misses the constraint with smaller budgets, however Bisection with Heuristic/Knapsack (green) and Bisection with Heuristic/Heuristic (red) took longer to meet r_{min}. Regarding the maximization of sampling rewards, results vary with Heuristic/Heuristic (magenta) working best at lower budgets, but then switches places with Heuristic/Heuristic Bisection (red) and eventually Heuristic/Knapsack Bisection (green). Interestingly, Bisection with Weighted Addition (blue) performed exactly the same as GPR (black), suggesting that the method does not work well with trying to maximize the sparsely distributed sampling rewards.

5.4 Conclusion

This chapter studied the BOOP where two reward functions were considered in orienteering on AGs. Because of the nature of multi-objective optimization, it was possible to define two sub-types of the BOOP, one attempting to maximize both objectives at once, and attempting to meet a minimum constraint on one objective while maximizing the other. Three methods were developed to solve the AGDM-BOOP - Heuristic/Heuristic which runs two instances of GPR in succession, Heuristic/Knapsack which runs GPR to form an initial path then finds ways to augment it using the 01-knapsack problem, and Weighted Addition which combines the two reward functions into a single function. Two methods were developed to solve the

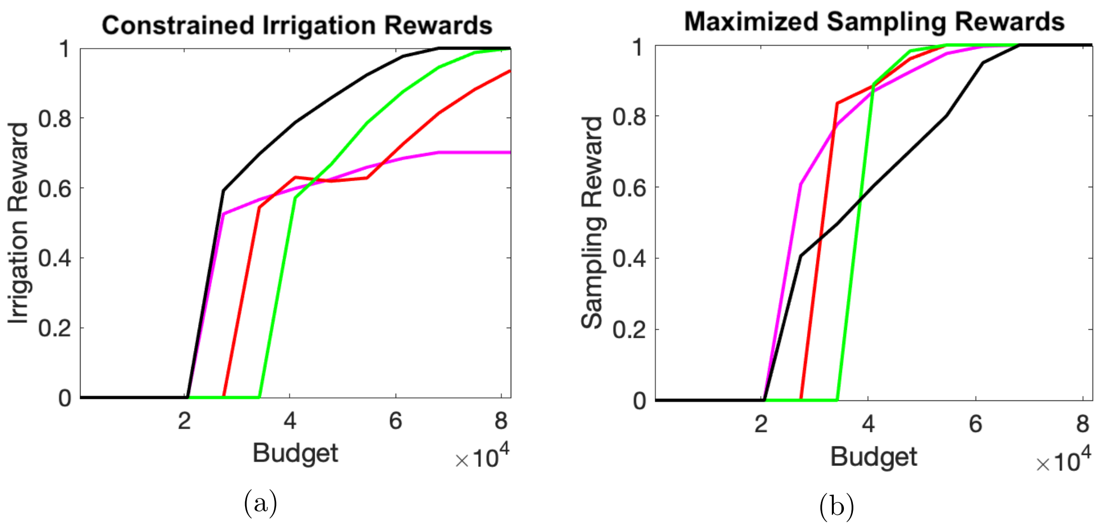


Figure 5.7: (a) Irrigation rewards R_2 and (b) sampling rewards R_1 for the Heuristic/Heuristic and Bisection AGOCBOOP methods, where $r_{min} = 0.5 \cdot \sum_{v \in V} R_2(v)$. **Legend**: Magenta is Heuristic/Heuristic, red is Heuristic/Heuristic Bisection, green is Heuristic/Knapsack Bisection, blue is Weighted Addition, and black is GPR. Note that lines for Weighed Addition and GPR are overlapping.

AGOCBOOP - Another Heuristic/Heuristic method which runs GPR until the constraint is satisfied and switches to maximization, and the Bisection GPR method which uses any AGDMBOOP method to dynamically search for the best input parameter that meets the constraint and maximizes the other reward. These methods were tested on a robotics problem concerning precision irrigation adjustment and soil moisture sampling within a real-world vineyard. Overall, the Weighted Addition approach seems to work best for the AGDMBOOP, while the Bisection method using Heuristic/Heuristic generally performed best on the AGOCBOOP.

Chapter 6

Stochastic Orienteering with Chance Constraints

The OP and its variants as formulated in previous chapters all rely on the assumption of deterministic values for the costs of edges. However, as discussed in section 2.6, problems based upon real-world situations are very much stochastic in nature. This chapter explores a method to solve the SOPCC using CMDPs and discusses some of its improvements. Work here was initially presented in [116, 117].

6.1 SOPCC Background

Orienteering, as defined earlier in section 3.1 assumes the cost function for edges is deterministic, i.e. $c : E \rightarrow \mathbb{R}_{\geq 0}$. For a given complete graph $G(V, E)$, a path $\mathcal{P}$ over G will have a total cost $C(\mathcal{P})$ which is the sum of all costs for each traversal of an edge. Because the cost function is deterministic, every potential path on G will have a deterministic cost. Therefore, it can be determined with certainty whether or not a path $\mathcal{P}$ violates a budget constraint B. In the deterministic OP, this fact means that the problem's solution (if it has one) is also deterministic (but not necessarily unique). To understand how this is relevant to the SOP, first the concept of a path policy must be explained. Then, the SOP and SOPCC can be defined.

6.1.1 Path Policy

Let $\mathcal{P}$ be a path in G and let $v_1 = \mathcal{P}(1)$, $v_1 = \mathcal{P}(1)$, $\ldots$, $v_{|\mathcal{P}|} = \mathcal{P}(|\mathcal{P}|)$ be the sequence of all $|\mathcal{P}|$ vertices along $\mathcal{P}$. For a path that lead from v_s to v_g, this means that $v_1 = v_s$ and $v_{|\mathcal{P}|} = v_g$. For $v_i \in \mathcal{P}$, the set of vertices following v_i in the path can be given as $\mathcal{S}(v_i) = \{v_{i+1}, v_{i+2}, \ldots, v_{|\mathcal{P}|}\}$. For convenience, the last vertex in the path follows the definition $\mathcal{S}(v_{|\mathcal{P}|}) = \emptyset$. Given a path $\mathcal{P}$, a path policy π is a function defined over $\mathcal{P} \times \mathbb{R}^+ \rightarrow \mathcal{P}$ such that for each $v_j \in \mathcal{P}$ and each $t \in \mathbb{R}^+$, the path policy is prescribed as $\pi(v_j, t) \in \mathcal{S}(v_j)$. In essence, for every t, $\pi(v_j, t)$ maps v_j onto one of the following vertices along the path.

For a simple path policy, defining a deterministic path along a predetermined sequence of vertices, the policy is simply given as $\pi(v_j = \mathcal{P}(j), t) \rightarrow \mathcal{P}(j+1)$ for all t and $1 \leq j < |\mathcal{P}| - 1$, and $\pi(v_{|\mathcal{P}|}, t) = \emptyset$ for all t. This means that any arbitrary path in G can be defined as a path policy. Therefore, the solution to the OP can be

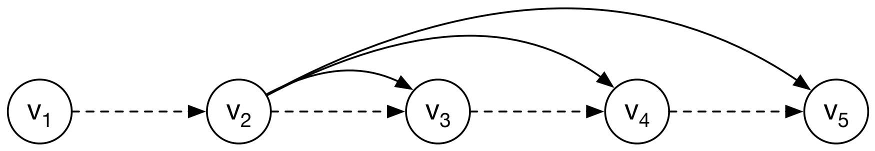


Figure 6.1: In a hypothetical path of 5 vertices, a path policy without shortcuts will lead it sequentially from v_1 to v_2, v_3, v_4 and finally v_5. However, if the path policy were allowed shortcuts, say at v_2, then the sequence of vertices visited may change. In particular, v_3, v_4, or both v_3 and v_4 may be skipped by an agent following a policy that tells it to do so, thus arriving at v_5 having visited fewer vertices.

given as a path policy as well. However, a path policy need not follow strictly the exact same ordering of vertices from a given path. In fact, as defined here a path policy can map a vertex in the path v_j to any successor vertex $S(v_j)$, depending on the given t. The reason to introduce path policies this way is to formalize the idea of taking a shortcut along a path solving an instance of the OP with random travel times along the edges. Assuming an agent starts moving along the path at time $t = 0$, the path policy introduces a formal way to skip some vertices along the way based on the current time and position. In particular, if the objective is to reach the last vertex before the temporal deadline B, a path policy π can be defined to skip vertices when the time t is approaching B (see Figure 6.1). Thus, a path policy allows an agent to take into account the stochastic of edge traversal when moving across a graph visiting vertices, and therefore is useful to define for the SOP.

6.1.2 The Stochastic Orienteering Problem

Let G, v_s, v_g, R, and B be defined as before. For every edge $e \in E$, let f_e be a probability density function (PDF) with positive support and finite expectation. Every time an edge $e_{i,j}$ is traversed, the incurred cost is not constant, but instead is a random variable $c_{i,j}$ whose PDF is f_e. For a path $\mathcal{P} = \{v_1, \ldots, v_{|\mathcal{P}|}\}$ and a path policy π, an agent starts at time $t = 0$ in vertex v_1 and moves to $v_i = \pi(v_1, 0)$ arriving at time t_i where t_i is a random variable with PDF f_{v_1,v_i}. Once in v_i, the agent moves to $v_j = \pi(v_i, t_i)$ arriving at time $t_i + t_j$, where t_j a random variable with PDF f_{v_i,v_j}. The process continues until the agent arrives at the last vertex of the path $v_{|\mathcal{P}|}$. In this case both the cost to complete the path and the reward collected along the path are random variables. In particular, the cost indicated by $C(\mathcal{P}, \pi)$, is the random variable of the cost associated with completing path $\mathcal{P}$ following policy π, and the reward indicated by $R(\mathcal{P}, \pi)$ is the random variable of the reward associated with with completing $\mathcal{P}$ following π. Now the SOP can be defined and is given as:

Stochastic Orienteering Problem (SOP): Given a graph $G(V, E)$ with stochastic cost function C, PDFs for the cost of each edge f_e, a reward function R, vertices within the graph $v_s, v_g \in V$, and a constant B, find a path $\mathcal{P} \subset V$ beginning at v_s and ending at v_g and policy π

which, for an agent following π, accrues an expected cost no more than B and maximizes the expected cumulative reward of visited vertices.

Notice here that the SOP requires maximizing the expected cumulative reward of the path $\max \mathbb{E}[R(\mathcal{P}, \pi)]$ while also limiting the expected total cost of the path $\mathbb{E}[C(\mathcal{P}, \pi)] \leq B$. For a given path and path policy it is always $\mathbb{E}[R(\mathcal{P}, \pi)] \leq R(\mathcal{P})$ because π can only skip vertices along the path and is thus constrained to being able to visit the same vertices. Considering that a SOP with all edges having zero variance PDFs is equivalent to the deterministic OP, the OP is evidently a special case of the SOP, and therefore the latter is NP-hard.

6.1.3 The Stochastic Orienteering Problem with Chance Constraints

The SOP as introduced seems useful for orienteering in environments where travel costs between vertices are stochastic, but it is not ideal for real-world problems. This is because solutions to the SOP only adhere to the budget constraint in expectation, meaning that it is very likely for the realized traversal cost of the path and policy to be greater than the budget. Violating the budget is a very undesirable situation, however, as this is often a restriction set by some physical constraint, such as fuel usage or time deadlines. Therefore, it makes sense to redefine the problem such that the probability of violating the budget constraint B is bounded by a pre-assigned constant $0 \leq P_f \leq 1$. This is the SOPCC, and it is formally defined as:

> **Stochastic Orienteering Problem with Chance Constraints (SOPCC):** Given a graph $G(V, E)$ with stochastic cost function C, PDFs for the cost of each edge f_e, a reward function R, vertices within the graph $v_s, v_g \in V$, and constants B, P_f, find a path $\mathcal{P} \subset V$ beginning at v_s and ending at v_g and policy π which, for an agent following π, accrues a total cost greater than B with probability no more than P_f, and maximizes the expected cumulative reward of visited vertices.

For a given failure probability P_f, the SOP asks to determine a path $\mathcal{P}$ and a path policy π that maximizes the expected sum of rewards $\mathbb{E}[R(\mathcal{P}, \pi)]$ such that $\Pr[C(\mathcal{P}, \pi) > B] \leq P_f$. As before, when the PDF for each edge has zero variance, the problem is equivalent to the deterministic version of the OP as long as $P_f < 1$ (when $P_f = 1$, the budget may be violated).

6.2 A Markovian Approach to Stochastic Orienteering

A common tool for constructing policies in stochastic decision making problems is the MDP and its derivatives. This section presents work from [116] focusing on finding path policies by constructing a special form of MDP and CMDP for the SOPCC.

6.2.1 MDPs for the SOP

For the time being, assume that a path $\mathcal{P}$ in G has been given. An agent moving along a path $\mathcal{P} = \{v_1, \ldots, v_{|\mathcal{P}|}\}$ and following a path policy π moves from vertex to vertex, and whenever it is positioned at vertex $v \in \mathcal{P}$ it can move to any of the subsequent vertices found in the set $\mathcal{S}(v)$. The time it takes to make this transition is characterized by the pdf associated with the edge it will traverse next. This naturally leads to a formalism based on the MDP over a suitably defined augmented state space. In the following, it is assumed that the reader is familiar with MDPs; [19] is referred for a comprehensive introduction. A finite MDP is given as $\mathcal{M} = \{S, A, \Pr, R\}$ where S is the set of states, A is the set of actions, Pr is the transition kernel, and R is the reward function associated with every state/action pair. Note that the notation for the reward function is the same as that used for the orienteering reward function in earlier chapters, and while they are separate concepts, the former is indeed derived from the latter. For the SOP defined in section 6.1, the MDP can be defined as follows:

- The state space is $S = V \times \mathbb{T}$, where V is the set of vertices in the path $\mathcal{P}$ and $\mathbb{T}$ is a suitable time discretization with step Δ. More precisely, $\mathbb{T}$ is a collection of successive time intervals where the k-th interval is $t_k = [k\Delta, (k+1)\Delta)$, assuming that $t_0 \in \mathbb{T}$ is the first one. An important question is how many intervals should there be in $\mathbb{T}$, and this will be answered later. The composite state (v_i, t_k) represents the fact that the agent arrived at vertex v_i during the time interval associated with t_k. In the following, for brevity, time t_k is used for the whole interval.

- The action set for each state (v, t) is $\mathcal{S}(v)$, i.e., the set of vertices following v along path $\mathcal{P}$. Note that the action set for (v, t) does not depend on t.

- The transition kernel Pr is the probability that the next state is (v_j, t_k), assuming that action a is executed from state (v, t_i). This is usually indicated as $\Pr((v, t_i), a, (v_j, t_k))$. Action a is an action in $\mathcal{S}(v)$, i.e., a vertex v_j following v along the path indicates that the agent will move from v to v_j. Therefore the transition probability is 0 for all states (v_i, t) with $i \neq j$. For states of the type (v_j, t_k) with $k \leq i$, the probability is also 0 because the agent can not go back in time. For the remaining vertices (v_j, t_k) with $k > i$, the probability is computed as

$$\Pr((v, t_i), v_j, (v_j, t_k)) = \int_{t_i\Delta}^{(t_i+1)\Delta} [F(\Delta(t_k + 1) - \xi) - F(\Delta t_k - \xi)] d\xi \qquad (6.1)$$

  where F is the cumulative function of the pdf f associated with the edge from v to v_j. For a given pdf, this integral can be computed numerically, off-line.

- The reward function R associated to the state/action pair $((v_i, t), v_j)$ is $R(v_i)$, i.e., the reward associated with the vertex v_i in G.

The cumulative reward function used to determine the policy is critical to the definition of an MDP, and therefore should be expounded. In this case a non-discounted reward is used, and to ensure that the overall reward remains bounded, two special states, the failure state s_f and the absorbing (or loop) state s_l, are added. The failure state also answers the formerly raised question of how many elements there should be in the discretized time set $\mathbb{T}$. Specifically, for a chosen discretization step Δ the number of intervals in $\mathbb{T}$ will be $N = \lceil \frac{B}{\Delta} \rceil$. The addition of s_f is used to represent the condition where the elapsed time is higher than the temporal budget B. The transition kernel Pr is accordingly extended so that for each vertex v_i, $\Pr((v_i, t_k), v_j, s_f)$ is the probability that v_j is not reached before exceeding B. The action set of the failure state s_f consists of a single action a_l leading to s_l with probability 1, while the action set of s_l consists of a single action, a_l, looping to itself with probability 1. The reward associated with these new state action pairs is $R(s_f, a_l) = R(s_l, a_l) = 0$, i.e., once entered those states do not accrue any more reward. Finally, for all states of the type $(v_{|\mathcal{P}|}, t_i)$ (recall that $v_{|\mathcal{P}|}$ is the last vertex along the path), an action a_l leading to s_l with probability 1 is added. Figure Figure 6.2 illustrates the structure of this MDP.

The structure of this MDP is similar to that presented in [45, 106]. By introducing suitable failure and loop states, it is possible to carefully design control policies that account for the possibility of failure. In this case, the failure probability is the probability of not reaching the last vertex along the path within the temporal deadline B, or equivalently, the probability of landing in s_f when following a policy. As usual, a policy for the MDP is a function $\pi : S \to A$ mapping states into actions, and in the proposed structure a policy for the MDP is indeed a path policy for the path $\mathcal{P}$. With the proposed MDP structure, the probability of reaching s_l under any policy π for the MDP is 1. Therefore, the following non-discounted reward function for a policy π and start state (v_0, t_0) can be considered:

$$\mathbb{E}[R(\mathcal{P}, \pi)] = R(\pi) = \mathbb{E}\left[\sum_{t=1}^{\infty} R(X_t, \pi(X_t))\right] \tag{6.2}$$

where X_i is the random variable for the state at time t and the expectation is taken with respect to the probability distribution induced by π. This expectation exists and is finite because the state s_l will be reached with probability 1 within a finite number of transitions and no more rewards will be collected from there onward.

6.2.2 CMDPs for the SOPCC

The MDP formulation focuses on a single objective function and is not suited to solving the SOPCC, where the goal is to maximize the expected collected rewards while making sure within a certain probability that the last vertex is reached within the budget. To achieve this goal, it is necessary to introduce a CMDP based approach, utilizing a model with more than one objective function where the policy aims at maximizing one of the objective functions while ensuring bounds in expectation for

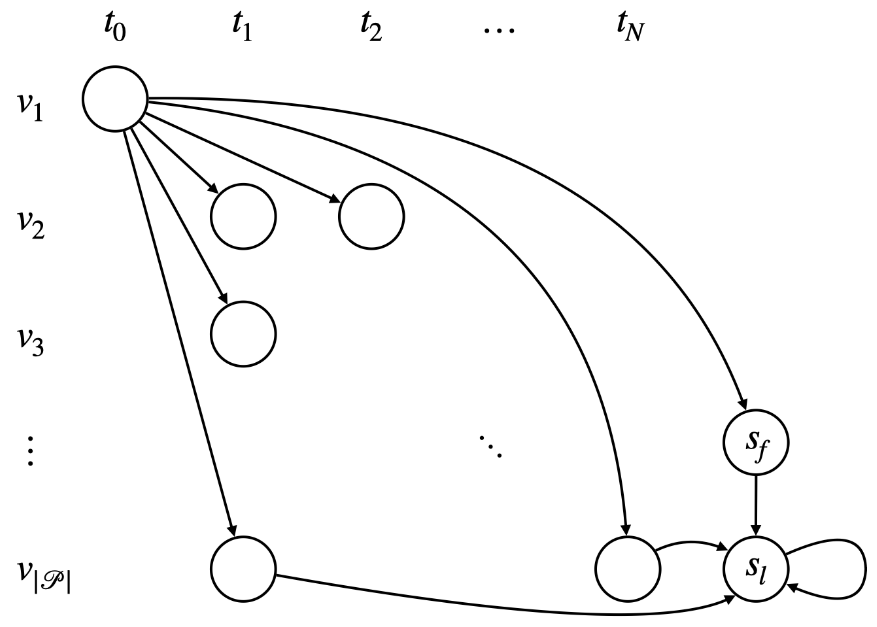


Figure 6.2: States in the MDP can be described as arranged on a grid with vertices (rows) and arrival times (columns). Arrows are depicted for some of the transitions with non zero probability. From a state (for example (v_1, t_0)) it is possible to go to any of the following vertices, and when moving towards a vertex (say v_2), the time of arrival can in principle be any of the times $t_i > t_0$ because of the random nature of the edge travel time. The event of reaching a vertex after the temporal deadline B has passed is modeled as a transition towards the failure state s_f. Note that once the state reaches the last vertex in the path $v_{|\mathcal{P}|}$, a deterministic transition is made to the loop state s_l where no more rewards are accrued.

the others [4]. More precisely, a new cost function $D(s, a)$ is introduced for every state/action pair $d : S \times A \to \mathcal{R}^+$ that is 0 everywhere except for (s_f, a_l) where it is 1. A CMDP defined this way, denoted by $\mathcal{CM} = \{S, A, \Pr, R, D, P_f, \beta\}$, can be solved through the following linear program, where the optimization variable ρ is defined over the set of state/action pairs $S \times A$, with β as a function that is 1 for the start state (v_1, t_0) and 0 everywhere else:

$$\max_{\rho} \sum_{(x,a)\in S\times A} \rho(x, a)r(x, a) \tag{6.3}$$

$$\text{s.t.} \sum_{(x,a)\in S\times A} \rho(x, a)D(x, a) \leq P_f \tag{6.4}$$

$$\sum_{y\in S} \sum_{a\in \mathcal{S}(y)} \rho(y, a)(\delta_x(y) - \mathcal{P}^a_{yx}) = \beta(x) \ \forall x \in S \setminus \{s_l\} \tag{6.5}$$

$$\rho(x, a) \geq 0 \quad \forall (x, a) \in S \times A \tag{6.6}$$

The linear program has a solution if and only if a policy π can be found that satisfies the constraint on the cost, and is uniquely defined by the solution vector ρ. The reader is referred to [40, 45, 106] for a detailed discussion about this approach. Theorem 6.2.1 shows that the above linear program indeed defines a policy satisfying the constraint on the failure probability P_f and is a minor adaptation of what was presented in [45]. Before stating the theorem it is useful to recall that the optimization variables $\rho(x, a)$ are so-called occupation measures and correspond to the following:

$$\rho(x, a) = \sum_{t=1}^{\infty} \Pr[X_t = x, A_t = a]$$

where X_t is the random variable for the state at time t and A_t is the random variable for the action at time t.

Theorem 6.2.1. *If the linear program admits a solution, then the associated policy π fails to reach the last state $v_{|\mathcal{P}|}$ within budget B with probability at most P_f.*

Proof. With reference to Figure 6.2, consider any path that starts from (v_1, t_0) and at a certain point enters s_f. Because cost D is zero everywhere, except in the state/action pair (s_f, a_l) and that in s_f just the action a_l can be taken, the constraint can be written as

$$\rho(s_f, a_l) = \sum_{t=1}^{\infty} \Pr[X_t = s_f] \leq P_f$$

and that is the probability of ever entering the failure state, i.e., the probability of exceeding the temporal deadline B. □

The formulation based on a CMDP leads to the following algorithm to solve an instance of the SOP introduced in section 6.1:

Algorithm 6.11 Solve SOPCC

```
Input: $G(V,E)$, $R$, $C$, $B$, $v_s$, $v_g$, $P_f$
Output: $\mathcal{P}, \pi$
1: $Z(e) = \mathbb{E}[C(e)] \forall e$
2: $\mathcal{P} \leftarrow$ OP-solver($G(V,E)$, $R$, $Z$, $B$, $v_s$, $v_g$)
3: $CM = \{\mathcal{P} \times \mathbb{T}, \mathcal{S}(v) \forall v \in \mathcal{P}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
4: $\pi, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
5: return $\mathcal{P}, \pi$
```

First, a new cost function for the edges is set up based on the expected value of every edge (line 1), so that the deterministic orienteering solver can be used to create an initial path (line 2). This is then used to set up and solve the CMDP $\mathcal{CM}$ as outlined above (lines 3-4). The quality of the solution of this proposed algorithm is dependent on the method used to solve the deterministic OP. For small problem instances one could obtain an exact solution using the standard mixed integer linear program to solve the OP (see section 3.1), and for larger problem instances a heuristic or an approximation method may be used. There are a number of problems with this approach, mainly the inefficient discretization of time in $\mathbb{T}$ and the lack of adaptivity with the vertices in $\mathcal{P}$. Accordingly, these are addressed in Chapter 7.

6.3 Complexity and Heuristics

The approach of adapting a deterministic orienteering path into a stochastic orienteering policy utilizing shortcuts to constrain failure probability faces a quick growth in computation time as the problem size increases. As the state space grows, the number of possible state-action pairs grows super-linearly and computation time becomes intractable. The approach proposed in this chapter makes use of a CMDP, which can be solved using a linear program and thus has a known computational complexity dependent on the number of state/action pairs $|\mathcal{S} \times \mathcal{A}|$. For the method described in section 6.2, this number is equal to

$$|\mathbb{T}| \cdot \frac{|\mathcal{P}|(|\mathcal{P}|-1)}{2} + 2 = \mathcal{O}(|\mathbb{T}| \cdot |\mathcal{P}|^2) \tag{6.7}$$

the number of time intervals for each vertex times the number of possible transitions in $\mathcal{P}$ (including shortcuts) plus the transitions to and from s_l. Since $|\mathcal{P}|$ determines how many possible transitions there are, this intuitively means that the computation time to solve the CMDP depends on two main factors, $|\mathcal{P}|$ and the chosen number of time intervals in $\mathbb{T}$. The natural methods to decrease the number of state/action pairs and therefore the computation time are to reduce the number of time intervals for each vertex and to reduce the length of the path $|\mathcal{P}|$. Doing one or both of these would likely have a very negative effect on the quality of the final solution, and are thus not preferred.

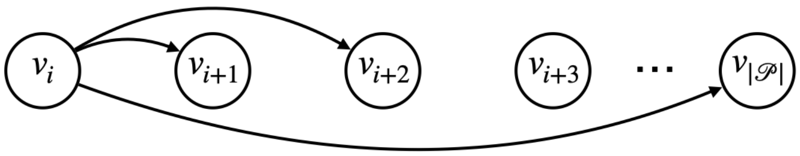


Figure 6.3: Connectivity from vertex v_i when $k_s = 3$.

There is another way to keep the CMDP computation time in check, without reducing the number of states. It is to limit number of state/action pairs given to the CMDP, which in turn limits the number of decision variables and therefore computation time. In section 6.1 the path policy was devised such that, at any given vertex $v_i \in \mathcal{P}$, the set of possible actions $\mathcal{S}(v_i)$ allowed for visiting any subsequent vertex $v_{i+1}, v_{i+2}, \ldots, v_{|\mathcal{P}|}$. As such, the total number of actions for all vertices is $\frac{|\mathcal{P}|(|\mathcal{P}|-1)}{2}$. To reduce the number of state/action pairs, instead of allowing the set of actions jumping to any subsequent vertex from v_i, only a size k_s subset of these actions is allowed, denoted as $A(v_i) \subset \mathcal{S}(v_i)$. Thus there will be only $k_s(|\mathcal{P}| - k_s) + \frac{k_s(k_s-1)}{2}$ actions for all vertices, which will significantly reduce the CMDP size if $k_s << |\mathcal{P}|$. The new number of state/action pairs will be

$$|\mathbb{T}| \cdot \left[k_s(|\mathcal{P}| - k_s) + \frac{k_s(k_s-1)}{2} \right] + 2 = \mathcal{O}(|\mathbb{T}| \cdot k_s |\mathcal{P}|) \tag{6.8}$$

Accordingly, the reduction in decision variables will also reduce the expected reward for a computed policy.

How the subset of $\mathcal{S}(v_i)$ is chosen will have a large effect on the expected reward of the resulting policy. A simple method of choosing $A(v_i)$ is to keep actions leading to the next k_s vertices, so $A(v_i) = v_{i+1}, \ldots, v_{i+k_s}$. With some combinations of paths and failure probabilities, this can lead to scenarios where there is no feasible solution. This is because an agent following a policy may be in a state where failure will occur because there is not enough budget left to travel to $v_{|\mathcal{P}|}$. A way to mitigate these scenarios is to keep the next $k_s - 1$ vertices and also $v_{|\mathcal{P}|}$ in $A(v_i)$, so that a shortcut to the end vertex can be taken when failure is imminent. An example of this vertex connectivity is shown in Figure Figure 6.3.

In an effort to reduce the loss in expected reward when eliminating potential shortcuts, the following heuristic was developed to determine the best shortcuts to keep at v_i:

$$H(v_i, v_j) = \frac{R(\mathcal{P}_{j,|\mathcal{P}|})}{\mathbb{E}[c_{v_i,v_j}]} \tag{6.9}$$

where $\mathcal{P}_{j,|\mathcal{P}|}$ is a sub-path of $\mathcal{P}$ starting at v_j and ending at $v_{|\mathcal{P}|}$. Recall that R and C are the reward and cost functions for a given path and c_{v_i,v_j} is a random variable for the cost to travel directly from v_i to v_j. This heuristic function weighs potential future reward against the expected cost for taking a given shortcut. The idea with this is that parts of $\mathcal{P}$ can be safely skipped if they add little to the cumulative reward and the cost for skipping them is small, thus the likelihood that a policy would skip them is large. Using the heuristic function, the set of actions $A(v_i)$ consists of the vertices v_j; $i < j < |\mathcal{P}|$ with the $k_s - 1$ largest values of $H(v_i, v_j)$ and $v_{|\mathcal{P}|}$.

Algorithm 6.12 Solve SOPCC with Heuristics

```
Input: $G(V, E)$, $R$, $C$, $B$, $v_s$, $v_g$, $P_f$, $k$, $s$
Output: $\mathcal{PT}, \pi_{new}$
1: $Z(e) = \mathbb{E}[C(e)] \forall e$
2: $\mathcal{P} \leftarrow$ OP-solver($G(V, E)$, $R$, $Z$, $B$, $v_s$, $v_g$)
3: for all $v_i \in \mathcal{P}$ do
4:     for all $v_j \in \mathcal{P}$ do
5:         $H(v_i, v_j) = R(\mathcal{P}_{j,|\mathcal{P}|})/\mathbb{E}[c_{v_i,v_j}]$
6:     In $\mathcal{S}(v_i)$ keep only $k_s - 1$ actions with highest $H$ and $v_g$
7: $CM = \{\mathcal{P} \times \mathbb{T}, \mathcal{S}(v) \forall v \in \mathcal{P}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
8: $\pi, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
9: return $\mathcal{P}, \pi$
```

Algorithm 6.12 shows pseudo-code making use of this heuristic. After the initial path is found (line 2) the heuristic value of actions for each vertex going to every other vertex is evaluated (lines 3-6). Of each action at a particular vertex v_i, only the top $k_s - 1$ actions are kept in $\mathcal{S}(v_i)$ as well as the action leading to the goal vertex (6). This reduced action set $\mathcal{S}$ for all vertices is then used to create the CMDP and a viable policy is found if one exists (lines 7-8).

6.4 Performance on Randomized Graphs

This section presents an overview of results obtained solving the SOPCC using the method described in this chapter. Synthetic SOPCC problems were designed to compare and contrast how changing the probability of failure and the number of shortcuts changes the expected reward collected by a policy.

6.4.1 Simulation Setup

Vertices of the graph G are obtained sampling the unit square with a uniform distribution, and edges are added to make a complete graph. Each vertex is associated with a constant reward randomly sampled from a uniform distribution over the interval $[0, 1]$. The stochastic travel time between vertices is obtained as follows. Let $d_{i,j}$ be the Euclidean distance between v_i and v_j, and $0 < \alpha < 1$. Then, the travel distance along edge (v_i, v_j) is

$$\alpha d_{i,j} + \mathcal{E}\left(\frac{1}{(1-\alpha)d_{i,j}}\right) \tag{6.10}$$

where $\mathcal{E}(\lambda)$ is a random sample obtained from an exponential distribution with parameter λ. As per the properties of the exponential distribution, it follows that the expected cost of the random variable associated with edge (v_i, v_j) is $d_{i,j}$ and the variance is $((1-\alpha)d_{i,j})^2$.

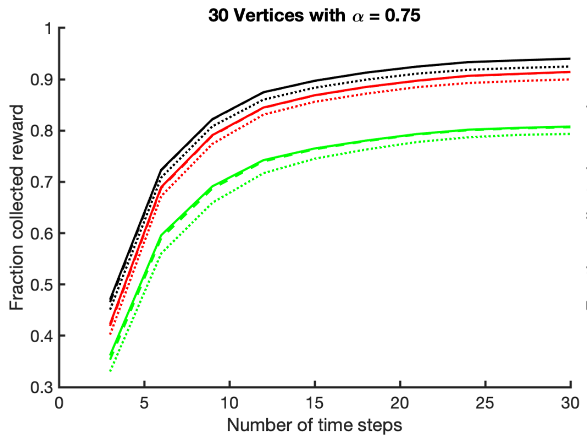
The method presented in this chapter utilizes an existing deterministic orienteering algorithm to compute the initial path. In every test displayed here, the S-algorithm heuristic described in [126] is used due to its relative speed and robustness. It is possible to use an exact solver based on a mixed integer program formulation, however this often takes longer than solving the CMDP and is impractical for the adaptive path method, which would repeatedly call the solver. Regardless, the output is a path $\mathcal{P}$ whose expected length is smaller than or equal to B. It is worth recalling, however, that if one were to follow all vertices in the path without using a path policy π, the temporal deadline B would be often missed, and for the setup described this happens roughly half the time.

The initial orienteering path $\mathcal{P}$ remains fixed for a given set of parameters across the methods introduced in this chapter so that a fair comparison can be made. The fraction of collected reward is $\frac{\mathbb{E}[R(\mathcal{P},\pi)]}{R(\mathcal{P})}$, or the expected reward collected by the policy divided by the total reward collected over the deterministic orienteering path. Afterward, the path is discarded, a new graph G is generated, and the processes is repeated until the methods have been compared over 10 instances. The results computed using a particular set of parameters are averaged for all paths.

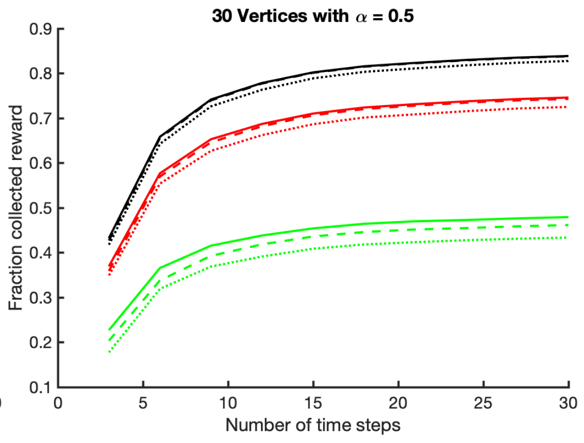
6.4.2 Varying Time and Length of $\mathcal{P}$

Algorithm 6.11, using the shortcut heuristic described in section 6.3 is analyzed with different values of k_s. When $k_s = \infty$, the heuristic is equivalent to keeping all shortcut actions. The number of time intervals in $\mathbb{T}$, the probability of failure P_f, and the parameter α are varied while keeping the length of $\mathcal{P}$ fixed, the results of which are shown in Figure 6.4. Also varied are the number of vertices in the path, which is shown in Figure 6.5, while keeping the number of time intervals fixed.

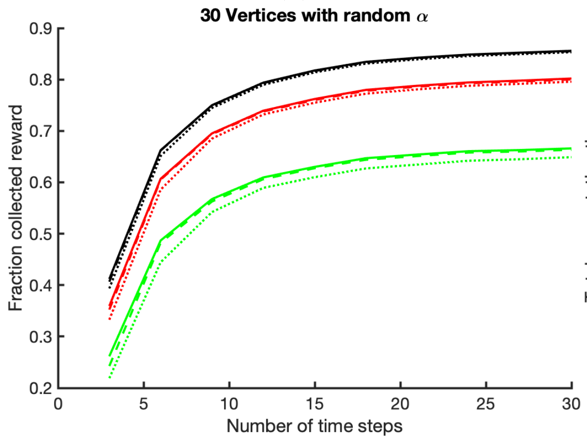
Several things are apparent in the results with a fixed initial path length $|\mathcal{P}| = 30$ and the number of time steps is varied. First, P_f has a large effect on the expected reward. The closer P_f gets to zero, the lower the expected reward. This happens because the policy requires taking shortcuts more often to meet the failure constraint, thereby skipping more vertices and missing out on reward. Second, as the number of time steps increases, the expected reward grows. This is due to the policy having a greater time resolution over which to choose actions; higher resolution means better choices are made. Third, the parameter α also has a large effect on the expected reward. Larger values of α mean less variance in the edge costs, and therefore each action directed by the policy is less risky so it can collect more reward. Fourth, even if α is random and different for each edge, the policy is still able to collect a large fraction of the reward despite $\mathcal{P}$ being fixed. This is because π will avoid taking edges with high variance when they do not yield high reward. Fifth, using shortcut heuristics has a small effect on the expected reward but a huge effect on the computation time. There is little expected reward loss when $k_s = 5$ but it consistently takes less than half



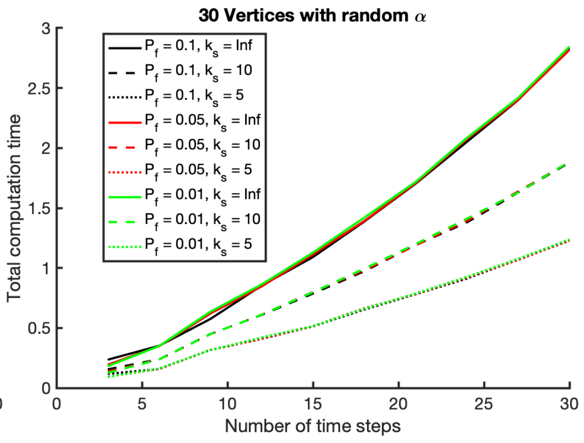
(a)



(b)



(c)



(d)

Figure 6.4: Rewards and computation time when the path length is fixed to 30 vertices. Each graph shares the same legend. (a) Average rewards for the fixed resolution method when $\alpha = 0.75$. (b) Average rewards for the fixed resolution method when $\alpha = 0.5$. (c) Average rewards for the fixed resolution method when α is a uniform random variable. (d) Average time taken to compute a policy π with the given parameters.

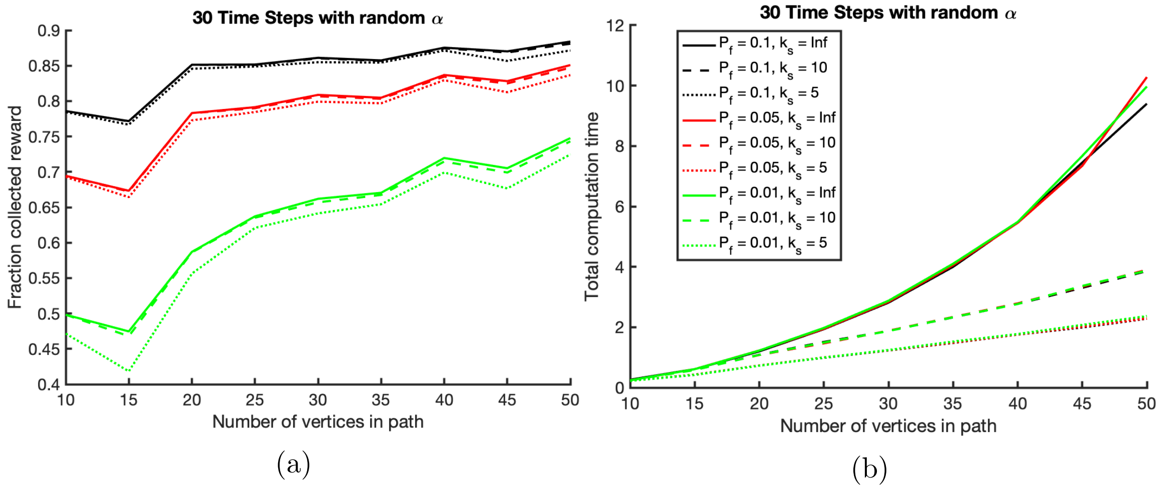


Figure 6.5: Rewards and computation time when the number of time steps is fixed at 30. Both graphs share the same legend. (a) Average rewards for the fixed resolution method when α is a uniform random variable. (b) Average time taken to compute a policy π with the given parameters.

the time of $k_s = \infty$ to run. Sixth, P_f and α have no effect on the run time, but the number of time intervals does. The fixed resolution algorithm will take roughly the same time regardless of the chosen failure probability or the variability in edge costs, but the computation time will increase with increased time steps roughly linearly. And finally, changes in α notably do not affect relative efficiencies when P_f or k_s are fixed, and only the spread changes.

When the number of time intervals is fixed $|\mathbb{T}| = 30$ and the length of the path is varied, a few more observations can be made. Changes in α effected these tests in a similar manner as before, whereby the only difference was a change in the spread of rewards collected but no change in the relative differences for each P_f or k_s. Results for a randomized α are representative of results for fixed values for α as well. The number of vertices in the path seems to slightly effect the policy's expected reward relative to the path. A longer $\mathcal{P}$ will lead to a policy that achieves a greater fraction expected reward than a shorter $\mathcal{P}$, and this trend seems more substantial with tighter failure bounds. Also, the number of vertices in $\mathcal{P}$ has a large effect on the amount of computation time required to create a policy, however shortcut heuristics are very helpful in limiting the growth rate of computation time.

6.5 Conclusion

When the assumption of deterministic edge costs is removed from the OP, finding effective solutions becomes challenging. The SOPCC is a formal variant of the OP using stochastic edge costs and requires a policy driven solution to efficiently collect rewards. This chapter studied how to utilize deterministic orienteering solutions to obtain path policies by formulating the problem as a CMDP. The state space was set up as a tuple containing vertices paired with time intervals corresponding to the time of arrival for the particular vertex. The action set allowed for traveling to any vertex further along in the path, but not to any other vertex in the graph. Transition probabilities for each state/action pair were defined by the PDF of the corresponding edge's cost, and a special failure state was created to account for any time that the sum of edge costs exceeded the budget. Since this method has a state space and action set that grows super-linearly, a heuristic was given to reduce the number of state/action pairs effecting the computation time. Finally, the method was tested on numerous randomized trials to evaluate how different parameters effect the quality of its solutions. Overall, this Markovian approach to solving the SOPCC gives useful policies that allow an agent to collect rewards over a graph while obeying a budget and probability of failure.

Chapter 7

Adaptive Approaches to Stochastic Orienteering

In Chapter 6, the SOPCC was introduced and a solution method to the problem was given using a deterministic OP solver and a specially defined CMDP. The approach described is, however, fairly limited in that it lacks adaptivity which inhibits its potential reward collection. This chapter tackles these limitations by discussing two ways in which the method can be improved. The first way involves choosing a nonuniform discretization of time when building the CMDP state space. The second way employs a flexible sequence of vertices to which a policy can direct an agent. A new algorithm utilizing these improvements is given along with heuristics to improve its time efficiency before validating with randomized experiments. Work here was first given in [117, 119].

7.1 The Discretization of Time

In this section, a refinement of the previous Algorithm 6.11 is proposed, which performs an adaptive discretization of the temporal dimension, as opposed to the uniform one discussed in section 6.2. The problem with a uniform time discretization approach is the state space $S = V \times \mathbb{T}$ is wasteful, as it uses the same resolution both for states (v, t) with very small probability of being reached as well as those with high probability. In regions where there are many states with high probability, a discretization with a smaller Δ should be used to determine policies, and in regions with low probability, fewer states are needed so a larger Δ is appropriate.

7.1.1 Sample Based Time Prediction

As the time to traverse an edge is a continuous random variable, it follows that the time a vertex is visited is a continuous random variable as well. Ideally, one would like to compute a policy of the type $\pi(v, t)$, for $t \in \mathbb{R}$. A continuous time policy can be approximated with a discretized policy, and the approximation becomes better as Δ shrinks. However, this increases the computation time. The idea with using an adaptive discretization is to allocate a more fine-grain time subdivision in high-density temporal regions and a more coarse one in low-density temporal regions. Here, a temporal region for a given vertex means the possible distribution of times when the vertex is reached. In order to improve the time discretization, a sample of arrival times for every vertex following the original path $\mathcal{P}$ is used to predict a good breakdown of $\mathbb{T}$. The algorithm is outlined in Algorithm 7.13.

Algorithm 7.13 Solve SOPCC with Adaptive Time

```
Input: $G(V,E)$, $R$, $C$, $B$, $v_s$, $v_g$, $P_f$, $k$, $s$
Output: $\mathcal{P}, \pi$
1: $Z(e) = \mathbb{E}[C(e)] \forall e$
2: $\mathcal{P} \leftarrow$ OP-solver($G(V,E)$, $R$, $Z$, $B$, $v_s$, $v_g$)
3: $\mathcal{K} \leftarrow$ simulation($G(V,E)$, $C$, $\mathcal{P}$, $k$)
4: $\mathbb{T} \leftarrow$ split(0, $B$, $\mathcal{K}$, $s$)
5: $CM = \{\mathcal{P} \times \mathbb{T}, \mathcal{S}(v) \forall v \in \mathcal{P}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
6: $\pi, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
7: return $\mathcal{P}, \pi$
```

The algorithm works much the same was as Algorithm 6.11. After finding an initial path (line 2), it is repeatedly executed k times without considering any policy, i.e., all vertices in $\mathcal{P}$ are sequentially traversed from v_1 to $v_{|\mathcal{P}|}$, without considering the temporal deadline B (line 3). During this process, for every vertex, the arrival time is logged in $\mathcal{K}$ and therefore it is possible to numerically approximate the temporal distribution of arrival times and their spreads (see top panel in Figure 7.1.) Then, these temporal distributions are used to build a tailored temporal discretization for each vertex in $\mathcal{P}$, splitting time into s equal probability intervals from $t = 0$ to $t = B$ independently for every vertex in the path. This makes a vertex-dependent temporal discretization $\mathbb{T}_v \in \mathbb{T}$. The number of intervals in this case is therefore $N = \lceil \frac{\mathcal{K} \leq B}{s} \rceil$ (Only samples where the arrival time is less than or equal to B are considered, since arrival times larger than B are encompassed by s_f). Two special segments are created at the beginning and the end, i.e., the first temporal segment in $\mathcal{T}_v$ starts at time 0, and the last one ends at time B.

7.1.2 Improving Adaptive Intervals

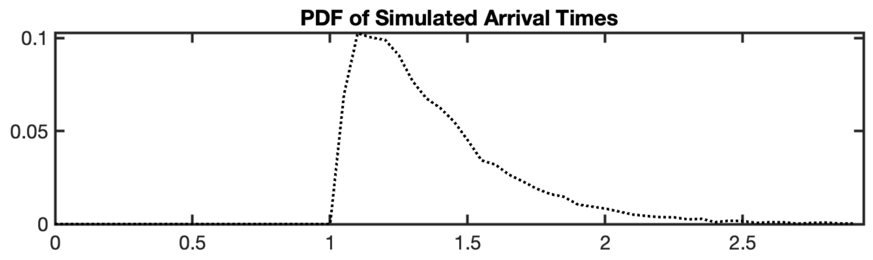
While it is useful to consider the distribution of arrival times to adaptively discretize the temporal dimension of the state space, this approach has a fatal flaw. It considers the arrival times produced by simulation the original path, rather than actual arrival times of the policy. Thus there can be instances where a state has a very coarse time discretization but the resulting policy directs an agent to this state with a high probability. For example, a policy might direct an agent to always skip a vertex, so the next vertex is visited with a time distribution that is much different than expected. Additionally, there can be value (in the form of increased expected reward) in a state space that limits the size of its largest time intervals, as smaller intervals are more accurate. There is no way of knowing what the distribution of arrival times will be for a certain policy without first computing the policy, therefore it makes sense to generalize our estimate by augmenting the simulated arrival times with evenly spaced data on the interval $0 < t < B$. This essentially combines the fixed discretization used in Algorithm 6.11 with the adaptive discretization used in Algorithm 7.13. Figure 7.1 shows an example of the uniform, adaptive, and combined discretizations. Once the set $\mathbb{T}_v$ is built for every vertex, the CMDP can be built as in the fixed-interval algorithm and solved.

An interesting aspect that emerges from using an adaptive discretization approach is that it is possible to reduce the number of elements in each of the sets $\mathbb{T}_v$ while essentially keeping the same performance as in the fixed discretization algorithm. Here, same performance means that in expectation the two algorithms collect the same reward while both ensuring that the probability of reaching $v_{|\mathcal{P}|}$ after B is less than P_f. However, the adaptive discretization algorithm is much faster because it has a much smaller state space.

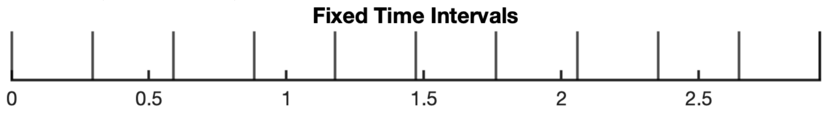
7.2 Adaptivity using Path Trees

A major pitfall of adapting a deterministic orienteering path into a stochastic orienteering policy using shortcuts is the lack of flexibility with vertices the policy can visit. The method described in section 6.2 develops a policy π that is capable of visiting only the vertices in $\mathcal{P}$. Thus, the maximum achievable reward is $R(\mathcal{P})$ and the expected reward can be much lower. However there are two situations which may arise where it is beneficial to deviate from the initial deterministic path. The first occurs when following π and arriving at a vertex v_i much earlier than expected. The second occurs when following π and arriving at a vertex v_i late enough to require taking a shortcut to some vertex beyond v_{i+1}. In both situations, there may be enough budget left that a new path $\mathcal{P}_{new}$ and policy π_{new} can be computed from v_i which will safely return an expected total reward larger than following the original policy from the current state to the goal vertex v_g. This method of determining a new policy during execution of the original policy is called online adaptation, and while the idea works well in these situations, it has the drawback of requiring an expensive computation to be done on-the-fly, which may take too long to compute $\mathcal{P}_{new}$ and π_{new} for use in real time.

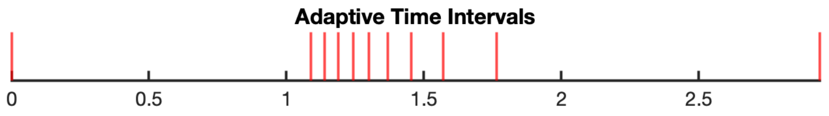
Instead of online adaptivity, $\mathcal{P}_{new}$ can be computed offline and cached before the execution of π. In this way, π_{new} will be a policy that considers a directed path tree, starting from v_1 and going to v_i along $\mathcal{P}$, then choosing to continue following $\mathcal{P}$ or following $\mathcal{P}_{new}$ until either branch reaches v_g. Since there may be many states where a new path should be computed, multiple branches can be added to the path tree. Note that it must be a directed tree, rather than a directed graph, because each vertex implicitly encodes information about what vertices have been previously visited (satisfying the Markov property), i.e. an agent at v_i has visited all vertices leading up to v_i from v_1 (disregarding shortcuts). Adding a new branch to the path tree increases the total number of vertices to $|\mathcal{P}| + |\mathcal{P}_{new}|$, and the resulting problem size is the same as one where there are no branches and the initial path $\mathcal{P}$ has a length of $|\mathcal{P}| + |\mathcal{P}_{new}|$. An example of a path tree is given in Figure 7.2.



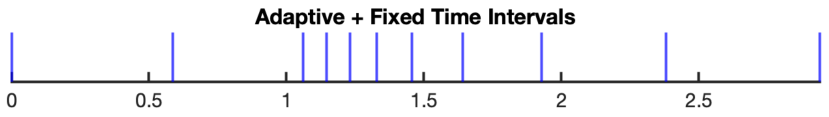
(a) Distribution of arrival times at a certain vertex v_i following strictly $\mathcal{P}$ ($k = 10000$).



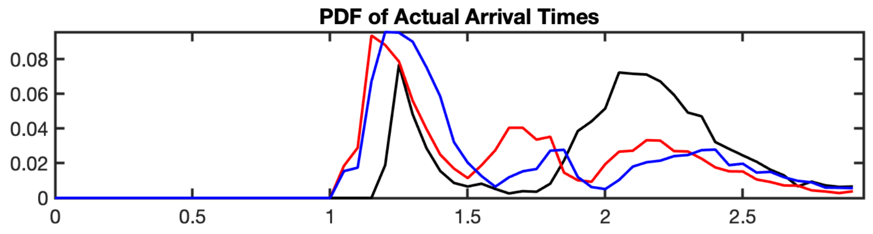
(b) Time discretization built by the algorithm using a fixed-interval approach with 10 intervals.



(c) Time discretization built using the adaptive-interval approach placing $s = 1000$ samples in each interval.



(d) Time discretization built by combining the adaptive-interval approach ($k = 10000$) with evenly spaced data ($k = 10000$) obtaining a total of $s = 2000$ samples in each interval.



(e) Distribution of arrival times at vertex v_i following policies computed with each type of time discretization. Note that each discretization has the same number of intervals.

Figure 7.1: A demonstration of the effect on vertex arrival times for different methods of discretizing time.

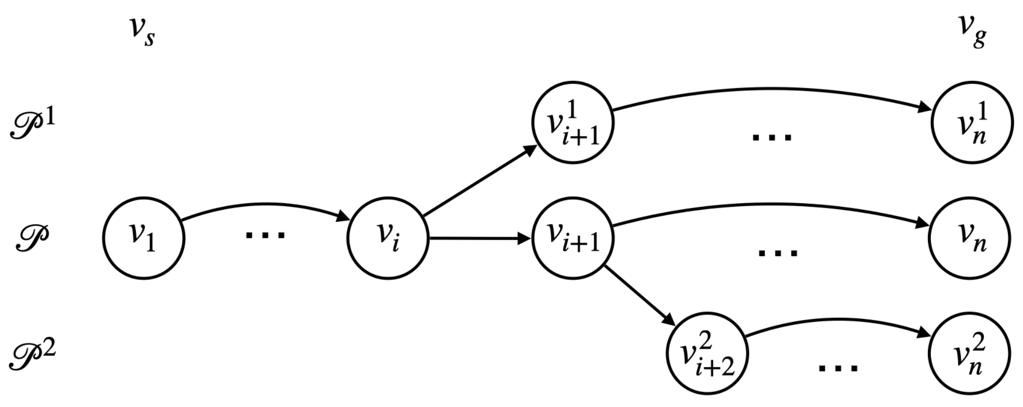


Figure 7.2: An example of a path with two additional branches at v_i and v_{i+1} to create a path tree. Note there is no inter-connectivity between branches.

Here, superscript notation is used to indicate a path starting at v_1 (or equivalently v_s) and ending at v_n (v_g) using a branch in $\mathcal{PT}$, where n is the total length of that particular path. As with the non-branching path $\mathcal{P}$, the policy may consider shortcuts that jump to vertices further along the path than the current vertex, but only along the same branch and any sub branches that exist. Using the example in Figure 7.2, v_1 may jump to any vertex in $\mathcal{P}$, $\mathcal{P}^1_{j>i}$, and $\mathcal{P}^2_{j>i+1}$, v_{i+1} may jump to only vertices further along in $\mathcal{P}$ and any vertex in $\mathcal{P}^2_{j>i+1}$, v^1_{i+1} can only jump to vertices further along in $\mathcal{P}^1$, etc.

Using this idea of creating a path tree to represent multiple possibilities for future vertex visitations leads to a new algorithm to solve an instance of the SOP:

Algorithm 7.14 Solve SOPCC with Path Tree

```
Input: $G(V,E)$, $R$, $C$, $B$, $v_s$, $v_g$, $P_f$
Output: $\mathcal{PT}, \pi_{new}$
 1: $Z(e) = \mathbb{E}[C(e)]\forall e$
 2: $\mathcal{P} \leftarrow$ OP-solver($G(V,E)$, $R$, $Z$, $B$, $v_s$, $v_g$)
 3: $CM = \{\mathcal{P} \times \mathbb{T}, \mathcal{S}(v)\forall v \in \mathcal{P}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
 4: $\pi, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
 5: $S_{jump} \leftarrow (v_i, t_j)$ where $\pi(v_i, t_j) > v_{i+1}$ for all $(v_i, t_j)$
 6: $\mathcal{PT} \leftarrow \mathcal{P}$
 7: for all $(v_i, t_j) \in S_{jump}$ do
 8:     $\mathcal{P}_{new} \leftarrow$ OP-solver($G(V,E)$, $R$, $Z$, $B - t_j$, $v_i$, $v_g$)
 9:     $\mathcal{PT} = \{\mathcal{PT}, \mathcal{P}_{new}\}$
10: $CM = \{\mathcal{PT} \times \mathbb{T}, \mathcal{S}(v)\forall v \in \mathcal{PT}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
11: $\pi_{new}, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
12: return $\mathcal{PT}, \pi_{new}$
```

Algorithm 7.14 uses the concept of a path tree to increase the expected reward. The idea is to compute an initial policy π using the method described in section 6.2 (lines 1-4), use that to determine where path branches should occur (line 5), and add new branches to the path tree (lines 6-9) to determine a new policy π_{new} (lines 10-11). There are some extenuating circumstances to consider when a newly computed branch is to be added to $\mathcal{PT}$. Some branches may follow a sequence of vertices that starts out the same as in the original path, i.e. $v_i, v_{i+1} \dots v_j$, and these can be shortened to start from v_j instead. Additionally, duplicates (both of branches and the original path) can often occur and these can be safely discarded. The final output of the algorithm is a policy π_{new} that determines which action to take for a sequence of vertices along $\mathcal{PT}$ at various times, maximizing the expected reward and bounding the failure probability to P_f, where the actions are possible vertex transitions (shortcuts) as outlined earlier. The expected reward $\mathbb{E}[R(\mathcal{PT}, \pi_{new})]$ will be at least equal to that from the method in Section section 6.2 and may even be greater than $R(\mathcal{P})$ due to the adaptive nature of the path tree.

7.3 Unified Method with Heuristics

In section 7.2 an adaptive path method was introduced that develops a path tree allowing variation in which vertices an agent following π_{new} is allowed to visit. The adaptive path algorithm is able to achieve a higher expected reward than the original method from section 6.2, however it comes at the cost of a significantly increased state space size. There may be many states where shortcuts are taken in the original policy, each requiring its own path branch, and as a consequence the number of vertices in $\mathcal{PT}$ can be many times greater than in $\mathcal{P}$. Additionally, without considering shortcut heuristics, the number of state/action pairs in the resulting CMDP grows much quicker than the non-adaptive path method. This number increases to:

$$|\mathbb{T}| \cdot \sum_{\mathcal{P}^b \in \mathcal{PT}} \left(\frac{|\mathcal{P}^b|(|\mathcal{P}^b| - 1)}{2} + 1 \right) = \mathcal{O}\left(|\mathbb{T}| \cdot |\mathcal{PT}|^2 \right) \tag{7.1}$$

where $|\mathcal{PT}|$ indicates the total number of vertices in the path tree. Since each new branch $\mathcal{P}^b$ adds a quantity of new vertices to $\mathcal{PT}$, this number can grow very quickly.

For each branch, the length of the resulting path $\mathcal{P}^b$ is variable. Because new branches are computed using a deterministic orienteering algorithm, the size of branches cannot be explicitly controlled. What can be controlled is the number of path branches added to $\mathcal{PT}$. To limit the number of branches, the number of states where a new path branch is computed should be reduced. An intuitive way of reducing this number is to consider adding path branches only when there is a high likelihood that they will be utilized. Since all non-loop states in $\mathcal{CM}$ can be visited only once, the occupancy measure vector ρ is equal to probability that each state/action pair will be executed. Therefore, the algorithm in section 7.2 can be modified such that only the k_b shortcut actions with the highest ρ values will be added to S_{jump}, where k_b is a user defined parameter. As such, there will be k_b branches added to the path tree instead of potentially $|S \times \mathbb{T}|$.

In order to achieve the best results on larger problems, all of the previously mentioned methods - Algorithm 6.12, Algorithm 7.13, and Algorithm 7.14 - should be combined. Because the number of state/action pairs in the CMDP grows very rapidly, the heuristics mentioned in this section are useful for problems of practical size ($|\mathcal{P}| > 15$). The pseudo-code for the entire process is shown in Algorithm 7.15.

Algorithm 7.15 Solve SOPCC with Adaptive Time and Path Tree

```
Input: $G(V, E)$, $R$, $C$, $B$, $v_s$, $v_g$, $P_f$, $k$, $s$
Output: $\mathcal{PT}, \pi_{new}$
 1: $Z(e) = \mathbb{E}[C(e)]\forall e$
 2: $\mathcal{P} \leftarrow$ OP-solver($G(V, E)$, $R$, $Z$, $B$, $v_s$, $v_g$)
 3: $\mathcal{K} \leftarrow$ simulation($G(V, E)$, $C$, $\mathcal{P}$, $k$)
 4: $\mathbb{T} \leftarrow$ split(0, $B$, $\mathcal{K}$, $s$)
 5: for all $v_i \in \mathcal{P}$ do
 6:     for all $v_j \in \mathcal{P}$ do
 7:         $H(v_i, v_j) = R(\mathcal{P}_{j,|\mathcal{P}|})/\mathbb{E}[c_{v_i,v_j}]$
 8:     In $\mathcal{S}(v_i)$ keep only $k_s - 1$ actions with highest $H$ and $v_g$
 9: $CM = \{\mathcal{P} \times \mathbb{T}, \mathcal{S}(v)\forall v \in \mathcal{P}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
10: $\pi, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
11: $S_{jump} \leftarrow (v_i, t_j)$ where $\pi(v_i, t_j) > v_{i+1}$ for all $(v_i, t_j)$
12: In $S_{jump}$ keep only $k_b$ states with highest $\rho$
13: $\mathcal{PT} \leftarrow \mathcal{P}$
14: for all $(v_i, t_j) \in S_{jump}$ do
15:     $\mathcal{P}_{new} \leftarrow$ OP-solver($G(V, E)$, $R$, $Z$, $B - t_j$, $v_i$, $v_g$)
16:     $\mathcal{PT} = \mathcal{PT}, \mathcal{P}_new$
17: $\mathcal{K} \leftarrow$ simulation($G(V, E)$, $C$, $\mathcal{PT}$, $k$)
18: $\mathbb{T} \leftarrow$ split(0, $B$, $\mathcal{K}$, $s$)
19: for all $v_i \in \mathcal{PT}$ do
20:     for all $v_j \in \mathcal{PT}$ do
21:         $H(v_i, v_j) = R(\mathcal{P}_{j,|\mathcal{P}|})/\mathbb{E}[c_{v_i,v_j}]$
22:     In $\mathcal{S}(v_i)$ keep only $k_s - 1$ actions with highest $H$ and $v_g$
23: $CM = \{\mathcal{PT} \times \mathbb{T}, \mathcal{S}(v)\forall v \in \mathcal{PT}, \Pr, R, D, P_f, \beta(v_s) = 1\}$
24: $\pi_{new}, \rho \leftarrow$ CMDP-solver($\mathcal{CM}$)
25: return $\mathcal{PT}, \pi_{new}$
```

The entire process essentially performs the method described in section 7.1 twice, once using $\mathcal{P}$ and again using $\mathcal{PT}$. Shortcut heuristics are used both times (lines 5-8,19-22) by calculating $H(v_i, v_j)$ for all pairs $i < j$ and the $k_s - 1$ largest valued actions are allowed into each CMDP $\mathcal{CM}$. It starts with the adaptive time method using the shortcut heuristic to produce a high-reward policy with a limited number of states that is ideal to build a path tree from (lines 1-10). A path tree is built according to the method in section 7.2 (lines 13-16), but with a limited number of branches (line 12) selected according to the most probable shortcut actions in π. The adaptive time method is then used again with a shortcut heuristic to produce the final policy π_{new} over $\mathcal{PT}$ (lines 17-24). This allows for efficient discretization of vertex-time space and limits the size of the CMDP (which is now much larger due to the vertex tree branches). All together, this combined method computes a new policy that is able to achieve a higher expected reward than any method separately on large problem sizes.

7.4 Evaluated Performance Improvements

As with Chapter 6, the methods described in this chapter are evaluated using a set of synthetic SOPCC problems based on a randomized graph. Vertices in G were sampled from the unit square with uniform distribution, and edges between them were given a random cost distribution based on a shifted exponential function. This is the exact same setup as in section 6.4. Again, the S-algorithm heuristic from [126] was used at all points where a solution to the OP was needed. In these tests, the fraction of expected collected reward at each data point is the average of 10 unique instances, as is the displayed run times. The only major difference between the results discussed here and those in section 6.4 is that the parameter α is not varied, and instead remains a uniform random number between 0 and 1 that is unique for each edge. The reason for not comparing each method on different fixed values of α is, as revealed earlier, that in each test it does not change the relative effectiveness of the prospective algorithms, only the spreads between them.

7.4.1 Adaptive Time Intervals

The set of tests here compared the fixed time interval algorithm to the adaptive time algorithm, using the two different sampling methods described in section 7.1. These results are discussed in [119]. Here, results in Figure 7.3 show the number of time steps being varied while the length of $\mathcal{P}$ is fixed, and results in Figure 7.4 show the length of $\mathcal{P}$ being varied while the number of time steps is fixed.

A few patterns emerge when comparing the adaptive time methods. The adaptive only method seems to fare favorably in relation to the fixed method when the number of time intervals is low, however it becomes worse than the fixed method with more time steps. This is because it does not provide enough flexibility for the policy due to the arrival time distribution being estimated from the deterministic path. The method combining adaptive and fixed time intervals is the clear winner in terms of expected reward collection. In almost every scenario, it gains visibly more reward than either the adaptive or fixed time intervals alone. The only exceptions happen when the number of time steps for each vertex is 3, which is where the adaptive only time intervals work best. It can be seen that the combined method reaches its potential much sooner than the others and the reward curve flattens out very quickly, suggesting that its use would allow cutting the size of $|\mathbb{T}|$ for more expedient policy calculations. The amount of time to compute a policy using the combined method does increase slightly, however it is more than made up for by the potential to decrease the number of time intervals needed to reach the same expected reward as the fixed method. An interesting trend also emerges with both the adaptive and combined methods, where as the number of vertices in $\mathcal{P}$ increases, the expected reward for the

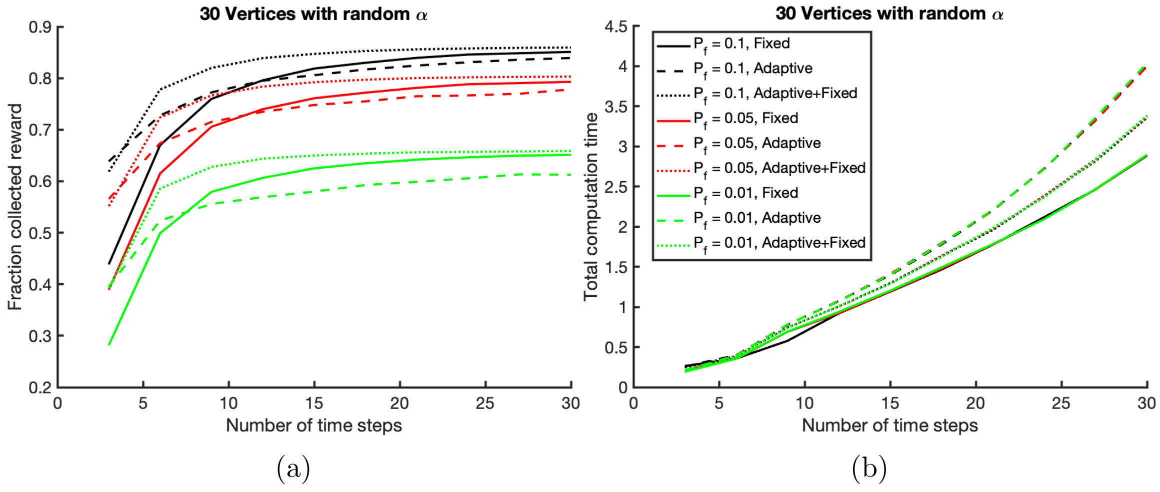


Figure 7.3: Rewards and computation time when the number of vertices in $\mathcal{P}$ is fixed at 30. Both graphs share the same legend. (a) Average rewards for the adaptive time method when α is a uniform random variable on the interval $(0, 1)$. (b) Average time taken to compute a policy π with the given parameters.

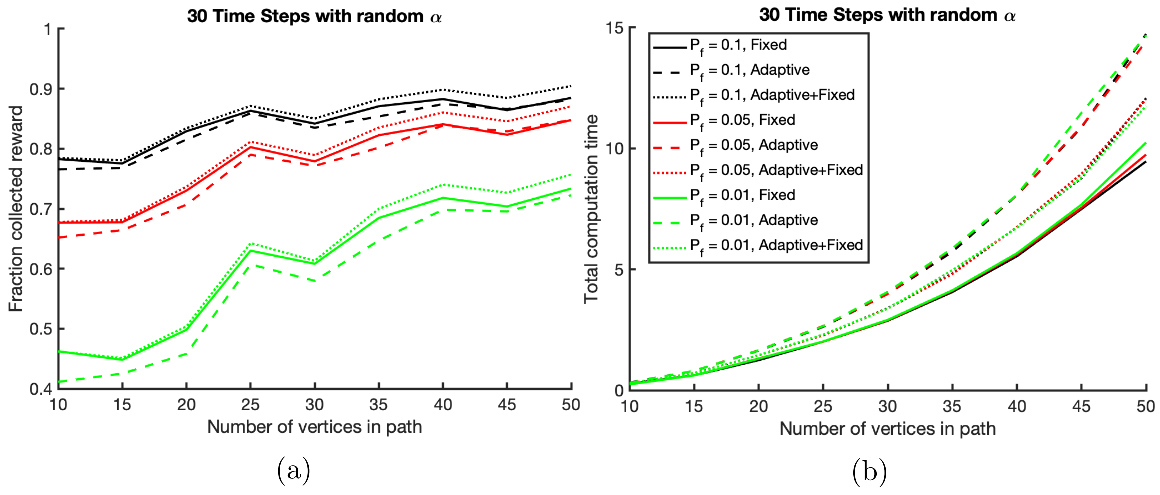


Figure 7.4: Rewards and computation time when the number of time intervals is fixed at 30. Both graphs share the same legend. (a) Average rewards for the adaptive time method when α is a uniform random variable. (b) Average time taken to compute a policy π with the given parameters.

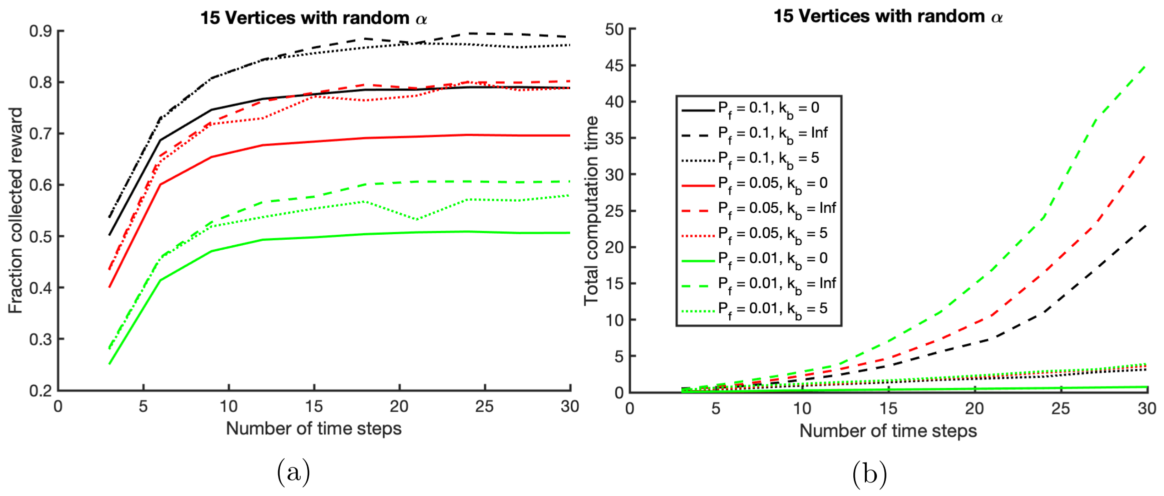


Figure 7.5: Rewards and computation time when the number of vertices in $\mathcal{P}$ is fixed at 15. Both graphs share the same legend. (a) Average rewards for the adaptive path method when α is a uniform random variable on the interval $(0, 1)$. (b) Average time taken to compute a policy π with the given parameters.

policies increases. This trend happens with the fixed interval method as well, but it is more pronounced with the two adaptive methods, and the reward gap widens. This shows how the usefulness of adaptive times becomes more prevalent as the problem size increases.

7.4.2 Adaptive Path Tree

The next set of tests was performed comparing the original SOPCC algorithm to the adaptive path algorithm described in section 7.2, utilizing the branch heuristics proposed in section 7.3. These results are originally discussed in [117]. Like before, results are shown where the number of time steps is varied and the length of $\mathcal{P}$ is fixed, as in Figure 7.5, and also where the length of $\mathcal{P}$ is variable and the number of time steps is fixed, as in Figure 7.6. It is worth repeating that $\mathcal{P}$ is the initial path, and that any $\mathcal{P}^b \in \mathcal{PT}$ can be much longer. For these tests, $k_b = \infty$ indicates all possible jump states are allowed to be branches and $k_b = 0$ means none are allowed (equivalent to Algorithm 6.11).

The benefits and detriments of the adaptive path algorithm are quite clear. There is a significant increase in expected reward collected compared to the original SOPCC algorithm for both $k_b = \infty$ and $k_b = 5$, showing that the method is very useful in maximizing reward when solving the SOP. Indeed, each additional branch can only increase the expected reward from the baseline established by the original SOPCC method ($k_b = 0$), because branching provides access to vertices previously unvisited by $\mathcal{P}$. There is also a significant increase in the computation time compared to the fixed resolution algorithm, by many times. This is true for both $k_b = \infty$ and $k_b = 5$, although the latter is much quicker than the former. It can seen that keeping k_b small

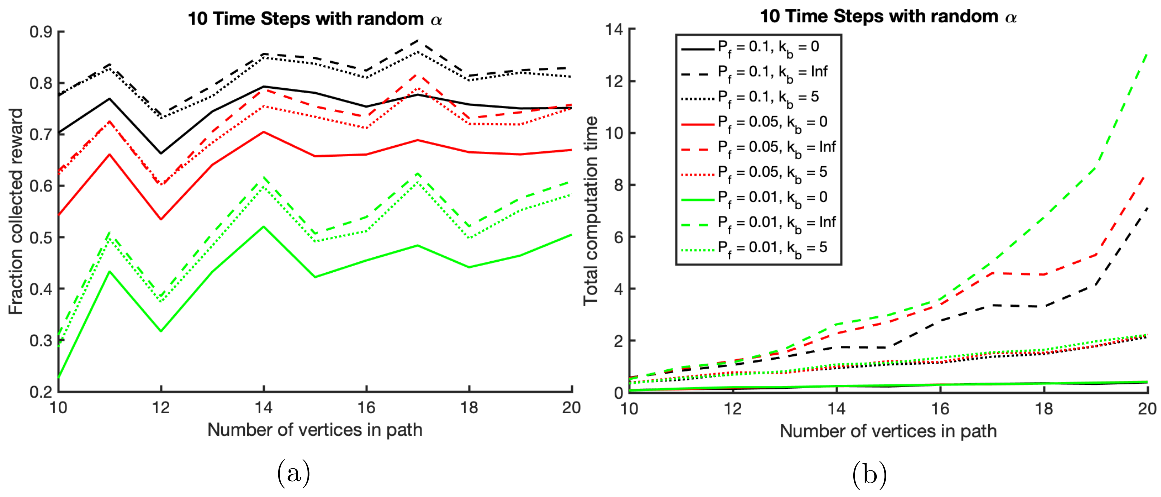
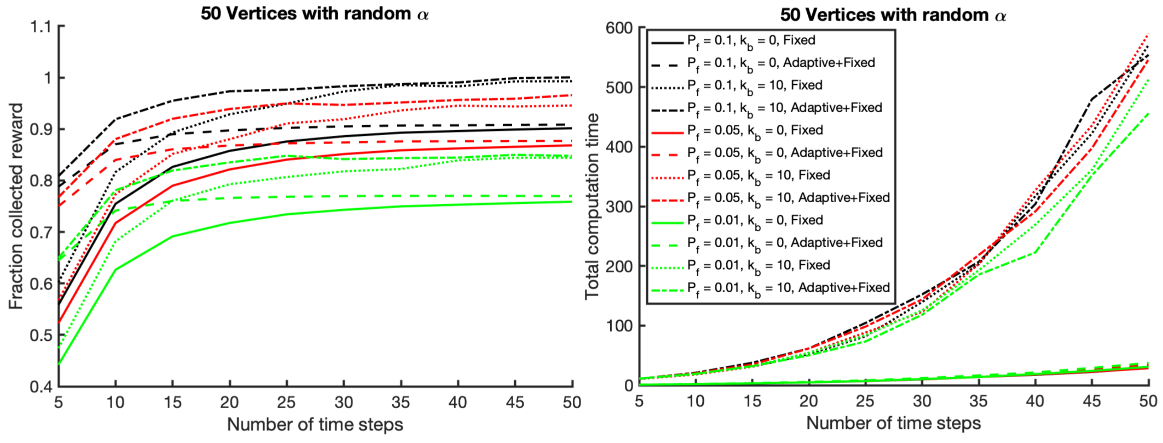


Figure 7.6: Rewards and computation time when the number of time intervals is fixed at 10. Both graphs share the same legend. (a) Average rewards for the adaptive path method when α is a uniform random variable. (b) Average time taken to compute a policy π with the given parameters.

helps control the total computation time while also obtaining most of the additional reward provided by the adaptive path algorithm. There is also a new pattern that emerges using this algorithm, which is the computation time depends on P_f when $k_b = \infty$. This is a natural consequence of the shortcut mechanism which each policy utilizes. As P_f gets smaller, π will direct more shortcut actions, and therefore the adaptive path method will add more branches to $\mathcal{PT}$. This is not a problem when k_b is bounded, however, because k_b limits the maximum number of branches utilizes. Finally, it should be pointed out that when $k_b = \infty$ the computation time scales very quickly with both the number of time steps and the length of $\mathcal{P}$, which means the full adaptive path algorithm is very limited in the size of problem it can solve.

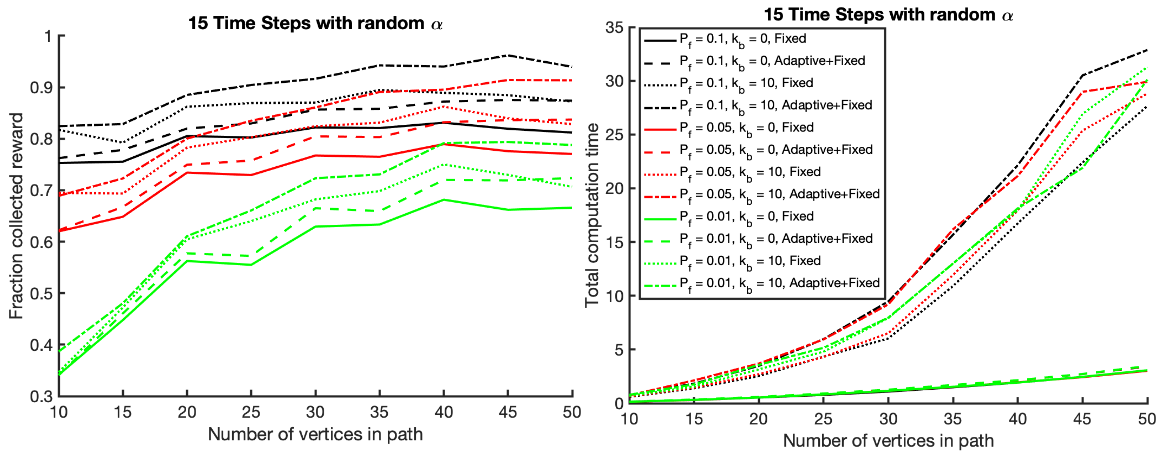
7.4.3 Combined Method

The last set of tests performed was done to compare the combined method from section 7.3, which combines the adaptive time and adaptive path methods and makes use of both types of heuristics. Results shown in Figure 7.7 and Figure 7.8 are the same as before; the time steps are varied while $|\mathcal{P}|$ is fixed, and $|\mathcal{P}|$ is varied while the time steps are fixed. Again, $k_b = 0$ means that no branching occurs. This is to directly compare the combined approach with the original SOPCC method and adaptive time interval approaches on the same set of $\mathcal{P}$. For this set of tests, whenever branching does occur, parameters are set at $k_b = 10$ and $k_s = 10$ to limit the size of the CMDP and get faster results. Because k_b was limited, longer initial paths could be studied, up to $|\mathcal{P}| = 50$. As shown earlier in this section, these heuristics obtain expected rewards very close to the methods without heuristics, allowing us to test problems with more vertices and branches than otherwise possible.



(a) (b)

Figure 7.7: Rewards and computation time when the number of vertices in $\mathcal{P}$ is fixed at 50. Both graphs share the same legend. (a) Average rewards for the combined method when α is a uniform random variable on the interval $(0, 1)$. (b) Average time taken to compute a policy π with the given parameters.



(a) (b)

Figure 7.8: Rewards and computation time when the number of time intervals is fixed at 15. Both graphs share the same legend. (a) Average rewards for the combined method when α is a uniform random variable. (b) Average time taken to compute a policy π with the given parameters.

The results are as expected. Combining the adaptive time interval approach with the adaptive path approach leads to the highest expected reward obtained for all values of P_f that were tested. Because of the use of adaptive time (combining adaptive and fixed intervals) in the combined method, the expected reward rises and levels out much quicker than using only the adaptive path method. The reverse is true as well; the use of adaptive paths in the combined method allows higher expected rewards. In some individual cases (not apparent on the charts because of averaging), the fraction of expected reward collected is greater than 1, meaning that π_{new} makes efficient use of branching to visit vertices not in the original path and collects even more reward. As for computation time, combining both methods does not significantly impact how long it takes to compute a solution. The combined method with heuristics shows only a slight increase in computation time compared to the adaptive path method with fixed time intervals. This increase in time can easily be compensated for by reducing the number of time steps for each vertex in $\mathbb{T}$, as the combined method consistently reaches its maximum expected reward using about half the number of time intervals it takes the adaptive path method.

7.5 Conclusion

Solving the SOPCC, as discussed in Chapter 6 requires the solution to be adaptive in the sense that it is policy driven. A path policy allows for this sort of adaptivity to occur such that vertices can be skipped when time is running low. However this method is limiting, especially when there is a small number of time intervals accounted for in the state space. section 7.1 showed ways to improve on the discretization of time, thereby allowing a policy to collect more expected reward with fewer time intervals. Additionally, section 7.2 showed how to improve on the initial path computed by a deterministic OP solver, such that newer policies can choose one of multiple routes in a path tree to maximize the expected reward. Both of these improvements to the original SOPCC method were tested on randomized graphs and shown to be superior, especially when combined together. Thus, a truly adaptive algorithm solving the SOPCC was developed.

Chapter 8

Large-Scale Stochastic Orienteering

This chapter discusses a way to improve the efficiency of solving the SOPCC on large-scale graphs, including AGs. In Chapter 6 and Chapter 7, path policy generation was notably limited to problems where the initial path contained between 30 and 50 vertices. Unfortunately, this is not to the scale of many real-world problems, even with heuristic improvements. A new supplemental concept, introduced in [116], is used to overcome this limitation.

8.1 Vertex Aggregation

The method of using CMDPs to compute optimal path policies for solutions to the SOPCC is rather limited in terms of the size of problem that can be solved, particularly due to practical restrictions on computational time and memory. Following the computation of an initial path using a deterministic OP solver, as sequenced by section 6.2 and section 7.2, the state space of the CMDP may become arbitrarily large as the sequence of vertices in $\mathcal{P}$ grows, making it intractable to solve. The concept of aggregation is utilized to overcome this limitation. Aggregation techniques used to reduce the size of CMDPs to a tractable level are not new, and an overview of them can be found in [20]. These techniques, however, usually employ a compression on the state space, meaning similar states are grouped together. This requires that special care be taken when computing transition probabilities between states, since aggregated states in general have unique transitions. Every part of the CMDP must be redefined in terms containing the new aggregate states, which is mathematically a difficult process.

Instead of aggregating on the state space, a technique which aggregates vertices before building the state space is proposed. This is where a set of sequential vertices in the initial path $\mathcal{P}(i,j) = \{v_i \dots v_j\}$ is aggregated into a single compound vertex which is representative of the sub-path between those vertices, denoted as $v_{i,j}$. Essentially, this is a way to represent a group of vertices as a single vertex, whereby the rewards and costs associated with that sequence are aggregated. Here, the reward for visiting a compound vertex is $R(v_{i,j}) = \sum_{k=i}^{j} R(v_k)$, and the cost for the traveling to a compound vertex from v_x (which is not part of the compound vertex) is $C(v_x, v_{i,j}) =$

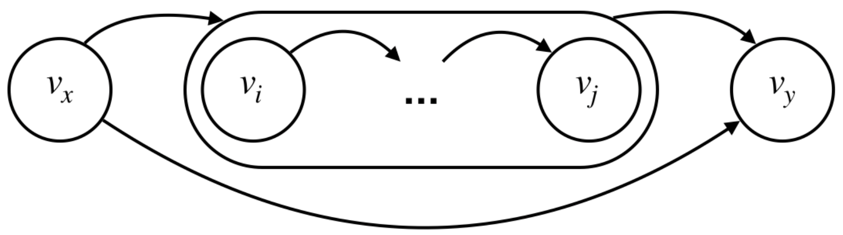


Figure 8.1: A compound vertex is an aggregate of multiple vertices. Here, v_x is any vertex (or another compound vertex) coming before $v_{i,j}$ in $\mathcal{P}$, and v_y is any coming after.

$C(v_x, v_i) + \sum_{k=i}^{j-1} C(v_k, v_{k+1})$. The cost of the compound vertex is defined as such because it has an explicitly defined entry vertex v_i and exit vertex v_j, and all other vertices between must be visited in sequence. Finally, this allows the path to be redefined as a sequence of vertices containing one or more compound vertices, i.e. $\mathcal{P} = \{v_1, \ldots, v_{i,j}, \ldots, v_{|\mathcal{P}|}\}$, and the total size of the state space for the resulting CMDP is reduced by $\mathbb{T} \times (j - i)$.

A compound vertex has special connectivity requirements regarding edges to other vertices in the path. Figure 8.1 shows how this works. All incoming edges to the compound vertex must enter at v_i and all outgoing edges from the compound vertex must leave at v_j. All individual vertices within the compound vertex are connected only in sequence as in $\mathcal{P}$, with no shortcuts allowed. This means that an agent leaving v_k can only go to v_{k+1} for all $i \leq k < j$ and an agent arriving at v_k must have left from v_{k-1} for all $i < k \leq j$. Essentially, when it comes to a path policy, a compound vertex is treated as a single vertex with no possible shortcuts in between and follows a pre-determined sequence. Additionally, this means that the rewards and costs associated with the compound vertex must be treated as a single unit. The reward for the compound vertex $R(v_{i,j})$ is not collected until all constituent vertices have been visited, i.e. the agent is at v_j, while the cost for visiting the compound vertex $C(v_{i,j})$ is enumerated immediately upon crossing edge $e_{x,i}$. In other words, it can be understood that the rewards for all aggregated vertices are "pushed" back to v_j as a single value, while the costs for all edges within the compound vertex are "pulled" forward to $e_{x,i}$ as a single random cost defined by the convolution of PDFs for the affiliated edges. An interesting consequence of this is that if an agent exhausts its budget while within $v_{i,j}$ before reaching v_j, no rewards $R(v_{i,j})$ are collected and all the costs $C(v_{i,j})$ are accrued.

Some advantageous aspects of compound vertices are that they may be of arbitrary size containing any number of vertices, they may be directly adjacent to other compound vertices, and each vertex within it may also be a compound vertex. These are a natural consequence of the way they are defined. In the case of nested compound vertices, where one or more are contained within a larger compound vertex, they admit a hierarchical structure that resolves to a singular compound vertex that just follows the sequence of all sub-compounds. As long as v_1 and $v_{|\mathcal{P}|}$ are not contained in any compound vertex, a suitable policy may be constructed for the SOPCC (if one exists).

8.2 Solving the SOPCC on AGs

With the introduction of vertex aggregation to solve the SOPCC on large-scale graphs, its is now reasonable to consider this problem in the context of AGs. A new type of problem can be defined as follows:

> **Aisle Graph Stochastic Orienteering Problem with Chance Constraints (AGSOPCC):** Given a graph $G(V, E) = AG(w, l)$ with stochastic cost function C, PDFs for the cost of each edge f_e, a reward function R, vertices within the graph $v_s, v_g \in V$, and constants B, P_f, find a path $\mathcal{P} \in V$ beginning at v_s and ending at v_g and policy π which, for an agent following π, accrues a total cost greater than B with probability no more than P_f, and maximizes the expected cumulative reward of visited vertices.

Since AGs are typically based on structured environments such as vineyards or warehouses, they typically contain far too many vertices for any SOPCC method from Chapter 6 or Chapter 7 to solve. Therefore, vertex aggregation is necessary for all but the smallest problem sizes. As discussed in section 3.2, AGs have a special structure which limits travel to vertices that are adjacent. Particularly, the vertices in an AG are connected by only two or three edges, depending on whether or not the vertex is within one of the graphs rows. These graphs may be made into complete graphs artificially, whereby new edges are added to connect every vertex to each other directly, however the new edges simply represent paths starting at one vertex and ending at another. The lack of organic connections between vertices makes paths built over AGs gives rise to a natural arrangement of vertices that can be aggregated together without reducing the number of possible shortcut opportunities.

The previous definition of an AG included a cost function that was not stochastic, and therefore the definition should be extended. An agent traveling across an edge in an AG would be subjected to a cost which is representative of the real-world space that is traversed. Additionally, there is an expectation that the agent, if representative of a vehicle such as a robot, has a defined maximum speed and therefore it has a minimum traversal time. It is also reasonable to assume that an agent usually does not take much longer than the minimum time, but on occasion does, and the extra time it consumes may be very large. Because of these considerations, the stochastic cost function is chosen to be a minimum cost α plus a random value drawn from an exponential distribution who's distribution depends on the minimum cost, i.e. $C(e) = \alpha + \mathcal{E}(1/(1 - \alpha))$ for all $e \in E$. Here, it is assumed that α is between 0 and 1, which is based on the reasonable assumption that all edges share an expected cost of 1 (in a deterministic problem, this would be equivalent to the CCAGOP with unit cost). It is set to 1 here only for convenience, however the expected cost for each edge can be any real positive value dependent on the problem at hand. Note that α is used in the same fashion as in section 6.4 when $d_{i,j} = 1$.

With a complete definition of a stochastic AG, the process of solving the AGSOPCC using vertex aggregation can be described. It follows the same steps as Algorithm 6.11 (or Algorithm 7.15 if using adaptive time or path improvements), however with a few key details unique for AGs. First, when an initial path is computed, it is preferable to use GPR instead of another heuristic (though if the graph is small enough an optimal solver may be used in its place). The design of GPR is to create a path containing a series of full-row traversals, where an agent moves from one side of the graph to the other side using one of the rows, and partial-row traversals, where an agent enters and exits a row from the same side while only visiting some of its vertices. The vertices visited in a full-row or partial-row must follow a specific sequence because of how they are connected with edges, and this makes them natural candidates for aggregation together as a single compound vertex. Thus, every full-row or partial-row can be aggregated into its own compound vertex without loss of optimality or changing the overall path. The cost of entering one of these compound vertices is $C(v_x, v_{i,j}) = k \cdot \alpha + \Gamma(k, 1-\alpha)$, where k is the number of vertices passed (including duplicates) along the full-row or partial-row and $\Gamma(k, 1-\alpha)$ is a random value taken from the two parameter gamma distribution. This cost, a shifted gamma distribution, is the result of the convolution of the PDF associated with each individual edge cost within the compound vertex.

With a set of compound vertices identified, it is possible to again aggregate these into larger compound vertices, resulting in a much smaller state space size than originally required. However, there is a unique consideration that needs to be addressed when aggregating compound vertices over AGs. A path over an AG requires that a full-row or partial-row be entered from the same side of the graph as the previous row was exited on. This means that edges connecting each compound vertex must also obey the same rule, and therefore possible shortcuts across the original path are limited. Figure 8.2 shows an example of how this restriction is handled in a relatively small graph $AG(w = 10, l = 8)$. If two or more compound vertices are aggregated together, the same rule applies but on a macro scale, meaning all edges leading to the new compound vertex must use the same side for entry as the first full-row or partial-row in the sequence, and all edges leaving must use the same side to exit as the last in the sequence.

After compound vertices are created shortcuts are properly defined, a CMDP can be used to solve the AGSOPCC without additional requirements. By and large, the use of compound vertices allows typically long paths over AGs to be aggregated into a few compound vertices that make policy computation much quicker. The example in Figure 8.2 starts with a path 53 vertices in length but is aggregated into 6 compound vertices plus the start and goal vertex, meaning more computational effort can be devoted to building a better policy using extra time intervals for the same effort it would take to build a worse policy with no aggregation and fewer time intervals.

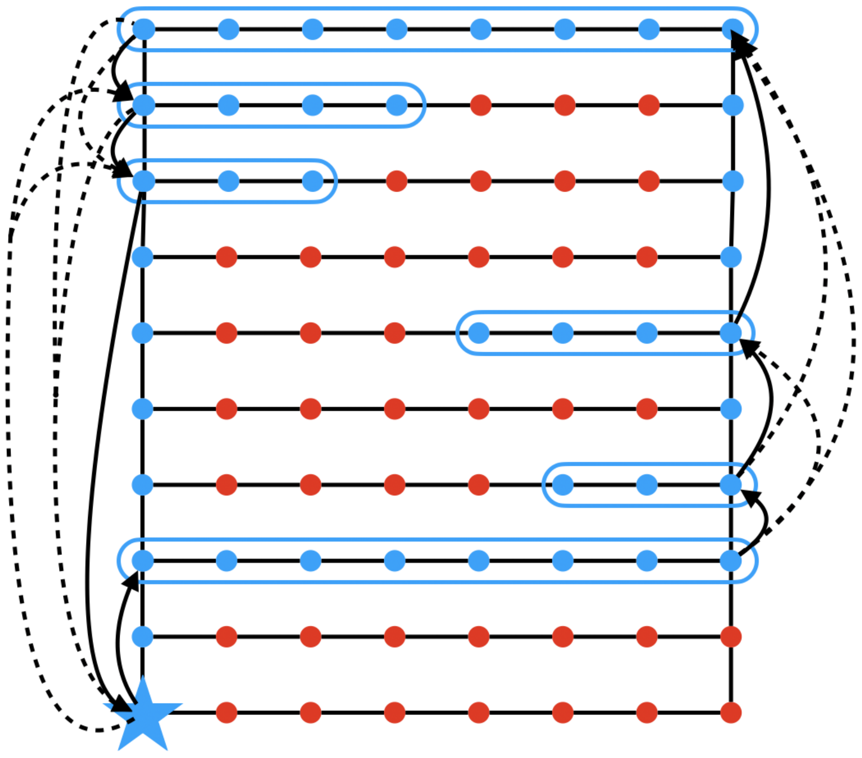


Figure 8.2: An example of the connectivity of multiple compound vertices for a path over an AG. Blue dots are vertices visited in the original orienteering path given by GPR, and are grouped together as compound vertices based on inclusion of a full-row or partial-row traversal. Black arrows represent the original sequence taken by the path, and dashed arrows represent possible shortcuts that allow for skipping compound vertices. The blue star represents both the start and goal vertex of the path.

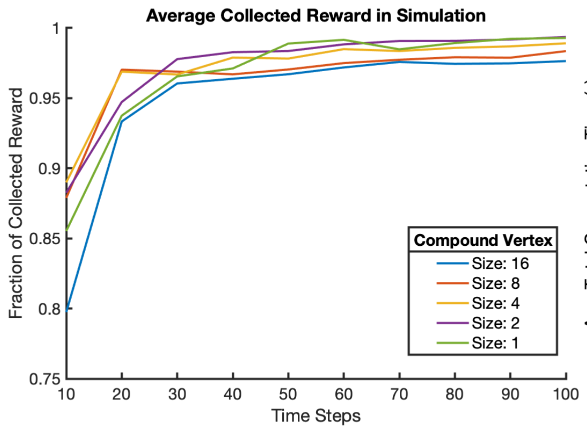
8.3 Performance and Efficiency Comparisons

To test the increase of computational efficiency on the SOPCC using using vertex aggregation, problems were designed around creating a path policy for a robot operation in a vineyard. An AG was created based on the same commercial vineyard used in previously in Chapter 5, and the same data was used for the reward of each vertex. The results displayed show averages of runs for all 9 datasets for every set of parameters used, with the corresponding fraction of collected reward showing the total expected reward collected divided by the total reward collected by the original path. The failure probability $P_f = \Pr[C(\mathcal{P}, \pi) > B]$ was fixed at 5% for the tests shown, however additional tests with different values of P_f were conducted that yielded similar results. α was fixed at 0.75 as well, and as discussed in section 6.4 changing this value did not result in substantially different outcomes. These results were originally discussed in [116].

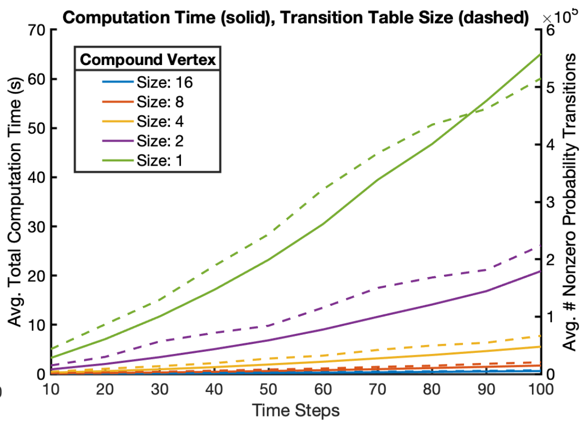
8.3.1 Non-Adaptive Path Results

This first set of results shows the problems with the parameters discussed solved using the non-adaptive version of the SOPCC solver from section 6.2. The budget B was fixed to 1/4 the traversable distance of the graph, generating initial paths using GPR with an average length of 14851 vertices. This was done because of time and memory constraints, as larger budgets mean longer initial paths. The paths consisted of an average of 76 full-rows and partial-rows, and therefore the initial number of compound vertices was 76 as well. These tests were conducted varying the amount of further aggregation used, with a size of 1 meaning no more aggregation (76 compound vertices), size 2 meaning two full-rows or partial-rows were aggregated together (38 compound vertices), and so on up to a size of 16 (average of 4.75 compound vertices). Increasing the size of aggregated compound vertices (thereby decreasing the number of compound vertices in the CMDP's state space) inevitably leads to an uneven distribution of compound vertices where some are larger than others. This is because the initial number of compound vertices does not evenly divide into the chosen size. This is not an issue and does not lead to a loss in performance.

Figure 8.3a demonstrates how varying the number of time steps $|\mathbb{T}|$ and the compound vertex size effects the amount of expected reward collected relative to the initial paths from GPR. Using large size compound vertices (meaning many full-rows and partial-rows aggregated together), performance noticeably deteriorates, but not substantially. Comparing a size of 1 with a size of 16 at $|\mathbb{T}| = 100$, there was only an average of 1.65% difference, meaning that the vast majority of expected rewards collected when using small sized compound vertices were also collected while using a large size. This shows that losses in performance are not significant between coarse and fine grain state spaces, at least in the vertex dimension. Throughout the entire range of time steps tested, using compound vertices of sizes 1, 2, and 4 resulted in very comparable rewards, and interestingly, a size of 1 was not always the top performer. This happens because the smaller granularity requires more accurate arrival



(a)



(b)

Figure 8.3: A comparison of the effect changing the size of each compound vertex has on the simulation's outcome when using the non-adaptive algorithm. (a) The reward collected over an AG for the SOPCC when full-rows and partial-rows are aggregated together into compound vertices, based on the number within each determined by the size parameter. (b) Solid lines: The average computation time needed when solving an instance of the AGSOPCC, when varying the compound vertex size and the number of time steps. Included is the time required to build the CMDP in Matlab and solve the CMDP in CPLEX. Dashed lines: Corresponding numbers of transitions in the state-action-state table of the CMDP where the probability of transition was greater than zero.

time estimations, which are dictated by the size of each time step Δ, and therefore the number of time steps. This is why more time steps result in a more predictable ordering regarding the fraction of reward expected. Also, increased vertex resolution can be a detriment when using fewer time intervals because state transitions are more likely to occur without crossing into new time intervals, thereby reducing their accuracy of estimation. This occurs using any size of compound vertex, however, and is more likely to happen when the ratio of time steps to compound vertices is small.

Figure 8.3b displays the average computation time of each CMDP and the average number of transitions in each CMDP with nonzero probability. It shows how the size of compound vertices used when building the state space egregiously effects the computation resources required to solve the problem. As expected from results in section 6.4, the number of time steps increases the amount of computation time needed super-linearly. More significantly, larger sized compound vertices greatly reduce the time needed to find a solution compared to smaller sized compound vertices. The significance of the change is two fold; not only does the size of the state space decrease because there are fewer compound vertices to consider, but so does the number of actions available to each compound vertex. This means that the number of state-action-state transitions with a nonzero probability changes greatly, which also greatly effects the number of decision variable in each CMDP, which is the largest determining factor in a CMDP's time to solution. This is why the time follows the number closely in the chart. Using a compound vertex of size 16 with 100 time steps completes in 0.91% of the time required when the compound vertex size is 1 with the same number of time steps. This is a 109 times speed increase. When comparing a compound vertex size of 4 instead, the difference changes to 8.53% or an 11.7 times speed increase. Clearly, the difference in computation time is substantial and justifies the minuscule loss in potential rewards when solving the AGSOPCC using the method outlined.

8.3.2 Adaptive Path Results

The second set of results show the simulations with the parameters discussed solved using the adaptive path tree version of the SOPCC solver from section 7.2, with the branch heuristics from section 6.3. For these tests, the budget B was fixed to 1/5 the traversable distance of the graph, resulting in initial paths from GPR with an average length of 11881 vertices, or 58 full-rows and partial-rows. Again, the amount of aggregation beyond the initial 58 compound vertices was varied, between 4, 8, and 16, in order to show how effective these extra aggregations are on the path tree model. Unfortunately, smaller aggregate sizes of 1 and 2 were not possible due to computation time constraints, as using the path tree method greatly increases the amount of time necessary to arrive at a policy, even when limited by k_b.

For the given parameters, the average fraction of expected reward collected can be seen in Figure 8.4a, and the average computation time can be seen in Figure 8.4b. Here, the number of time steps, the size of each compound vertex, and the number of allowed branches are changed. The results shown are not surprising given what

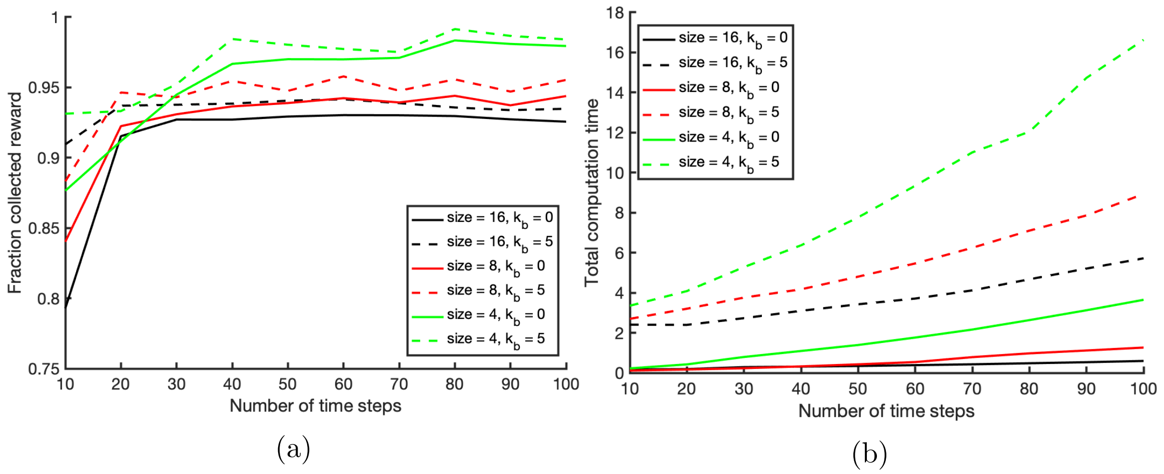


Figure 8.4: A comparison of the effect changing the size of each compound vertex has on the simulation's outcome when using the adaptive path algorithm. (a) The reward collected over an AG for the SOPCC using the path tree method, when full-rows and partial-rows are aggregated together into compound vertices, based on the number within each determined by the size parameter. (b) The average computation time needed when solving an instance of the AGSOPCC using the path tree method, when varying the compound vertex size and the number of time steps. The time shown is the total time required to build the CMDP in Matlab and solve the resulting CMDP in CPLEX.

was discovered using compound vertices of different sizes in the previous set of tests. The largest fraction of expected collectible rewards comes when using the smallest compound aggregation size, and the smallest fraction when using the largest aggregate size. There was, however, a noticeable leveling off in the reward after increasing the number of time steps to 30 for sizes 16 and 8, and 40 time steps for size 4. This suggests that after a certain point, the number of time steps no longer effects the reward outcome and it is therefore unnecessary to increase $|\mathbb{T}|$ beyond that. Finally, the change in k_b from 0 (no path branching) to 5 (up to 5 path branches) shows a small increase in the expected reward collected, but also a large increase in the amount of computation time needed. Looking at $|\mathbb{T}| = 40$ and $size = 4$, there was an increase of only 1.8% in the expected reward collected, however there was a 5.3 second (or a 583%) increase in the average amount of time needed to find a solution to the AGSOPCC. Unfortunately, the large increase in computation time necessary meant that k_b had to be limited to 5, as a larger branching factor would have necessitated much more time to solve for a minuscule reward boost. These results do, however, show promise in the regard that the path tree method is still effective at increasing the potential rewards of a AGSOPCC, even when vertex aggregation is used, though trade-offs must be considered.

8.4 Conclusion

The SOPCC, when solved using any of the methods described in Chapter 6 and Chapter 7, requires quite a bit of computational resources to arrive at a solution, even on small graphs with small budgets. In problems with graphs of moderate size or moderate budgets, the amount of time required increases dramatically, and therefore the concept of vertex aggregation was introduced to mitigate the increase in state space size and number of actions causing the time surge. Vertex aggregation allows the initial path built for a SOPCC to be condensed into a representation that has much fewer compound vertices (so-called because they each represent a group of vertices) within it, thereby reducing the computational need of the resulting CMDP. They are also useful when deployed on the AGSOPCC, as they allow for a compact description of each full-row or partial-row in a path over an AG. Because of this, it is possible to solve the AGSOPCC on graphs of realistic size containing tens of thousands of vertices, such as those relevant to vineyard robotics. Therefore, the use of compound vertices made using vertex aggregation described in this chapter is a powerful tool for stochastic orienteering on large-scale graphs.

Chapter 9

The Lagrangian Method for Stochastic Orienteering

The method described in section 6.2 and all the subsequent modifications to it, rely on the use of a specially built CMDP to solve a SOPCC. The typical approach to optimal policy generation for CMDPs is to use a linear program, as explained in section 2.5. This proceedure, however, is very time consuming to complete, especially as the size of the CMDP grows. This chapter discusses the Lagrangian method for solving CMDPs as a way to mitigate some of the time grown for finding solutions to SOPCC CMDPs. The work here was first presented in [118].

9.1 Acyclic Absorbing CMDPs

The Lagrangian method from [4], which uses dynamic programming as opposed too linear programming for CMDPs, has the potential to speed up computation of policies (see [79] for a discussion on how dynamic programming might be more efficient). This section develops a basis for using the Lagrangian approach to solve CMDPs specifically built for the SOPCC, and discusses some properties of these CMDPs that will be exploited later.

9.1.1 Absorbing CMDPs

A realization of the execution of a policy π over a single-constraint CMDP will induce a trajectory consisting of a sequence of states and resulting in the collection of a reward for each state $R(s, \pi(s))$ as well as the accumulation of a cost for each state/action pair utilized $D(s, \pi(s)), \ldots, D(s, \pi(s))$. In literature, multiple types of CMDPs are commonly investigated, however two are prevalent and discussed here, and are used because they ensure that the sum collected rewards is bounded under any possible trajectory. The first type is the discounted infinite horizon CMDP, which allows for an infinite number of executions of the CMDP. Here, the trajectory is the sequence of states $s_1, s_2, \ldots s_\infty$, the total collected reward is given by $\sum_{n=1}^{+\infty} \gamma^n R(s_n, \pi(s_n))$, and the total accumulated cost is given by $\sum_{n=1}^{+\infty} \gamma^n D(s_n, \pi(s_n))$. The discount factor γ is usually given as a constant on the interval $[0, 1)$ to prevent the total reward from tending to infinity during calculation. Note how γ is used in both the reward and cost functions. In some formulations, γ is not used in the calculation of accumulated costs, since these values are usually meant to be constrained in totality (unlike reward, which is unconstrained). The second type is the finite horizon CMDP, which limits the execution of the CMDP to only N transitions, where N is a given constant. Here, the trajectory is the sequence of states $s_1, \dots, s_N$, the total collected reward is given by $\sum_{n=1}^{N-1} R(s_n, \pi(s_n))$, and the total accumulated cost is given by $\sum_{n=1}^{N-1} D(s_n, \pi(s_n))$.

For some problems, the assumptions made for finite and infinite horizon CMDPs are either impractical or unnecessary. The type of CMDP developed in section 6.2 is an absorbing CMDP, which does not utilize these assumptions. A CMDP is absorbing if it has a special state $s_l \in S$ with a single action a_l such that:

- For every possible policy π, the trajectory will eventually enter s_l with a probability of 1.
- The reward for this state and action is $R(s_l, a_l) = 0$.
- The cost for this state and action is $D(s_l, a_l) = 0$.
- The transition kernel dictates $\Pr(s_l, a_l, s_l) = 1$.

The absorbing state s_l ascribes a special property to the CMDP which ensures that every policy is terminal, i.e. the trajectory eventually stops accumulating rewards and accruing costs. Therefore, the total reward still has finite expectation without needing a discount factor or limiting the trajectory horizon. This implies that the amount of transitions in a trajectory is a finite random number, meaning that the total collected reward and total accumulated cost for a trajectory can be calculated in the same way as the finite horizon CMDP, where N is the number of states in the full length of the trajectory.

9.1.2 Acyclic CMDPs

While absorbing CMDPs necessarily have finite length trajectories, there is no requirement for a trajectory to be of reasonable length. Indeed, a trajectory may be of arbitrary length as long as it is guaranteed to eventually reach the absorbing state. This includes trajectory lengths which are longer than the size of the state space $|S|$. The reason why this is possible is that in general, a CMDP may contain a set of state/action pairs which induce a cycle in the CMDP graph, leading to a repetition of states in some induced trajectory. For many problems, this behavior is an integral part of the system dynamics. However, there are situations in which this behavior does not make sense. One common example of these situations is when the state space contains a time component. Since there is no way to reverse the flow of time, a transition to some state in the past should not be possible. Likewise, the flow of time is continuous and unchanging, meaning that even near instantaneous transitions still advance the time component of the state. For these types of situations, it is impossible for states to repeatedly appear in a trajectory of the CMDP if the state space contains time as a component. The CMDP used for SOPCCs, as given in section 6.2 is designed in this way, using time as one of the two dimensions of its state space.

A CMDP which does not contain actions that allow states to be visited more than once is called an acyclic CMDP. In order for a CMDP to be considered acyclic, it must meet the following conditions:

- For any two states s_i and s_j, if s_j is reachable from s_i, then s_i is not reachable from s_j.
- Any action a leading from state s back to s must have $R(s, a) = 0$ and $D(s, a) = 0$.

The second condition allows for the inclusion of absorbing states, which do not induce cycles in the graph that effect the total reward or cost. A special property of an acyclic CMDP (and acyclic directed graphs in general) is that it has a topological ordering, an arrangement of states that is consistent for every trajectory. Formally, it means that for every action $a \in A$ leading from state s to s', s' occurs later in the ordering than s. For example, for a CMDP with a topological ordering $s_i > \cdots > s_j$, all trajectories containing both s_i and s_j will have s_i occurring before s_j regardless of states in between.

For a generic acyclic CMDP, there can be multiple possible terminal states (if any at all) in an acyclic CMDP and therefore they are not generally absorbing. However, the properties of absorbing CMDPs are desirable for computation, and it is advantageous to convert an acyclic CMDP into one that is also absorbing. This can be accomplished simply by adding an absorbing state s_l to the CMDP and giving all terminal states (states with no outgoing actions) the action a_l. Taking a_l from these states should gather no additional reward or costs and result in arriving at s_l with probability 1. This achieves a CMDP with no loops and a single terminal state, termed here as an acyclic absorbing CMDP (aaCMDP). Of note is that the CMDP used to find path policies for the SOPCC is set up in this way, with a designated absorbing state and corresponding action that locks progression in a non-accumulating loop. Therefore, it matches properties described here can be characterized as an aaCMDP.

9.2 Sequential Stochastic Decision Making

Dynamic programming is commonly used to solve MDPs, with methods like Value Iteration (VI) and Policy Iteration (PI), however it is not possible to apply these methods directly to CMDPs. VI works iteratively with repeated sweeps through the state space, updating a value function defined over each state until convergence is obtained, i.e. the value function remains static across iterations or the maximum difference of any state between consecutive iterations is less than a preassigned threshold θ. Then, the optimal policy can be extracted from the value function. PI is similar to VI, however the policy is updated at every iteration and convergence happens when the policy no longer changes. In order to apply the dynamic programming principle to CMDPs, one must apply the Lagrangian approach which converts the CMDP to an MDP using Lagrangian multipliers before a policy can be produced.

9.2.1 The Lagrangian Method

For a CMDP with J constraints, $\lambda \in \mathbb{R}^J$ is a real non-negative vector representing the Lagrangian multiplier. A new reward function $R^\lambda(\pi)$ can be introduced for a given λ and π

$$R^\lambda(\pi) = R(\pi) - \sum_{j=1}^{J} \lambda_j (D_j(\pi) - U_j) \tag{9.1}$$

which defines the Lagrangian reward with respect to the CMDP's J cost functions D_j and constraints U_j. Shown by theorem 9.9 in [4], a policy π^* is optimal for $\mathcal{CM}$ if and only if

$$R(\pi^*) = \sup_\lambda \max \left[R(\pi) - \sum_{j=1}^{J} \lambda_j (D_j(\pi) - U_j) \right] \tag{9.2}$$

For the choice of λ used, an associated MDP can be constructed and solved with VI or PI. This Lagrangian approach, however, has a major downside in that the appropriate λ needed for a particular CMDP must be found numerically.

The CMDP design used in the process of solving a SOPCC given by section 6.2 has only 1 cost function rather than J, and therefore the appropriate Lagrangian multiplier λ in this case is a scalar. Because of this, the reward function can be written slightly differently as

$$R^\lambda(s, a) = (1 - \lambda) R(s, a) - \lambda D(s, a) \tag{9.3}$$

parameterized by $\lambda \in [0, 1]$. Written like this, the reward function is no longer infinite as $\lambda \leftarrow \infty$ but bound between $[-D(s, a), R(s, a)]$. Typical Lagrangian methods used in literature (see [55]) suggest searching for the appropriate λ between 0 and a "sufficiently large" value, however it is generally not obvious where the second value lies and could be unreasonably large. Instead, λ as used above acts as a weight parameter, the optimal value of which can searched for within a defined interval rather than being unconstrained. When $\lambda = 0$, the reward function R^λ results in an MDP maximizing R without consideration of the cost function. When $\lambda = 1$, the opposite is true and R^λ considers only the cost function D. Importantly, this means that as λ increases towards 1 the reward for all state/action pairs $R^\lambda(s, a)$ is strictly decreases, and the cost function of the associated MDP decreases as well.

The following theorem, a rewritten version from [4], establishes the relationship between the policy found for two Lagrangian MDPs and the optimal stochastic policy of the CMDP on which they are based:

Theorem 9.2.1. *The optimal policy π^* for a CMDP with a single constraint is a randomized mixture of two deterministic policies of its Lagrangian MDP, i.e.*

$$\pi^* = \sigma\pi^*_{\lambda_1} + (1-\sigma)\pi^*_{\lambda_2} \tag{9.4}$$

where $\pi^*_{\lambda_i}$ is the optimal policy for the Lagrangian CMDP for λ_i and σ is a suitable mixing parameter to be determined.

This theorem does not state that any mixture of two deterministic policies for the Lagrangian MDP is the optimal policy for the CMDP, but rather states two such policies exist which will mix to form the optimal stochastic policy given an appropriate value for σ. As will be shown later, a λ can be chosen for which the resulting policy mixture π^* obeys the constraint $\mathbb{E}[D(\pi^*)] \leq P_f$, as long as such a policy exists for the CMDP.

9.2.2 Finding λ

The Lagrangian reward function $R^\lambda(s, a)$ is a weighted sum of the reward and cost functions of the given CMDP taking λ as a parameter. In order to obtain a policy satisfying the constraint $\mathbb{E}[D(\pi^*)] \leq P_f$ while also maximizing $R(s, a)$, the optimal value for λ needs to be determined. Because λ was defined within a closed interval, it is possible to utilize a bracketed root-finding algorithm to approximate the optimal value. Algorithm 9.16 is one way of finding λ, which makes use of the Bisection Search method to iteratively shorten the interval in which the optimal value is known to lay until an approximation error on the reward is satisfied.

The precision of the numerical approximation of λ can be controlled with two parameters. The first parameter, ε, provides a maximum bound on the absolute difference in reward for the two policies π_{hi} and π_{lo} which are computed with the current bounds of λ using λ_{hi} and λ_{lo}. Since the expectation of total reward r is known to monotonically decrease as λ increases, it can be reasoned that all Lagrangian policies computed with $\lambda_{lo} < \lambda < \lambda_{hi}$ have expected total reward falling somewhere between r_{hi} and r_{lo}. Thus, the maximum difference in expected reward between the optimal policy and λ is less than or equal to ε. The second parameter, θ, provides a stopping condition on Algorithm 9.16 that causes it to terminate execution when the interval for λ is sufficiently small. This is a necessary parameter because of a it is not guaranteed that $|r_{hi} - r_{lo}|$ will ever be less than or equal to ε. The reason for this is that the expected total reward $r(\beta)$ from following a policy π is not differentiable with respect to λ. That is, $r(\beta)$ is a step function over changes in λ. This is because there are only a finite number of deterministic policies for a given MDP, and for sufficiently small changes in λ the optimal policy does not always change. It is possible that the optimal λ for the Lagrangian MDP lies on the edge of a step where λ_{hi} and λ_{lo} approach from opposite sides and $|r_{hi} - r_{lo}|$ is always greater than ε until $\lambda_{hi} = \lambda_{lo}$. Algorithm 9.16 has a worst case computational complexity of $\mathcal{O}(\log(\frac{1}{\varepsilon}) \cdot X)$, where X is the complexity of VI. The use of VI may be replaced by PI, however an updated version of VI will be given soon.

Algorithm 9.16 Lagrange Bisection

```
Input: $P$, $R$, $D$, $P_f$, $\beta$, $\varepsilon$, $\theta$
Output: $r_{hi}$, $r_{lo}$, $d_{hi}$, $d_{lo}$, $\pi_{hi}$, $\pi_{lo}$
 1: $\lambda_{hi} \leftarrow 1$; $\lambda_{lo} \leftarrow 0$
 2: $r^{\lambda}, r, d, \pi_{hi}$ =VI($P$, $D$, $R$, $D$, $S$)
 3: $r_{hi} \leftarrow r(\beta)$
 4: $r^{\lambda}, r, d, \pi_{lo}$ =VI($P$, $D$, $R$, $D$, $S$)
 5: $r_{lo} \leftarrow r(\beta)$
 6: $\lambda = 0.5$
 7: while $|r_{hi} - r_{lo}| > \varepsilon)$ and $(|\lambda_{hi} - \lambda_{lo}| > \theta)$ do
 8:     $r^{\lambda}(s,a) \leftarrow (1-\lambda)R(s,a) - \lambda D(s,a); \forall (s,a) \in S \times A$
 9:     $r^{\lambda}, r, d, \pi$ =VI($P$, $R^{\lambda}$, $R$, $D$, $S$)
10:     if $d(\beta) > P_f$ then
11:         $\lambda_{lo} \leftarrow \lambda$
12:         $r_{lo} \leftarrow r(\beta)$
13:         $d_{lo} \leftarrow d(\beta)$
14:         $\pi_{lo} \leftarrow \pi$
15:     else
16:         $\lambda_{hi} \leftarrow \lambda$
17:         $r_{hi} \leftarrow r(\beta)$
18:         $d_{hi} \leftarrow d(\beta)$
19:         $\pi_{hi} \leftarrow \pi$
20:     $\lambda \leftarrow (\lambda_{lo} + \lambda_{hi})/2$
21: return $r_{hi}$, $r_{lo}$, $d_{hi}$, $d_{lo}$, $\pi_{hi}$, $\pi_{lo}$
```

There exists more bracketed root-finding algorithms that are capable of numerically approximating the optimal value of λ, however Bisection Search is one of the most well-known. Another possible method to use is the Illinois False Position Search method from [46]. Algorithm 9.16 can easily be modified to use this method instead simply by swapping 20 for the following formula:

$$\lambda \leftarrow \frac{0.5(\lambda_{lo}(d_{lo} - P_f)) - \lambda_{hi}(d_{hi} - P_f)}{0.5(d_{hi} - P_f) - (d_{lo} - P_f)} \tag{9.5}$$

The benefit of using Illinois False Position Search over using Bisection Search is that it usually converges faster. However, this is not guaranteed. Nevertheless, it provides on average a significant speed improvement and therefore is included here for plenum.

Theorem 9.2.1 states that two deterministic policies for the Lagrangian MDP with different values of λ can be mixed together to obtain the optimal stochastic policy for a CMDP, but it leaves unexplained how to find the appropriate mixing parameter σ. A closed formula for σ exists (see [80] page 139), which requires that the constraint $\mathbb{E}[C(\pi^*)] \leq P_f$ must be active for π^*, leading to the following computation for the mixing parameter:

$$\sigma = \frac{P_f - d_{lo}}{d_{hi} - d_{lo}} \tag{9.6}$$

Algorithm 9.17 shows how to use σ to mix the two policies π_{hi} and π_{lo} together, obtaining an optimal policy for the CMDP in $\mathcal{O}(S^2)$ time. The result of using this in conjunction with Algorithm 9.16 is a policy π for a CMDP that satisfies the constraint $\mathbb{E}[C(\pi^*)] \leq P_f$ and maximizes the expected total reward $R(\beta)$ to within ε of the optimum.

Algorithm 9.17 Policy Mixture

```
Input: $\pi_{lo}$, $\pi_{hi}$, $\sigma$
Output: $\pi$
1: for all $s \in S$ do
2:     for all $s \in A(s)$ do
3:         $\pi(s,a) \leftarrow \sigma\pi_{lo}(s,a) + (1-\sigma)\pi_{hi}(s,a)$
4: return $\pi$
```

9.2.3 Exploiting aaCMDPs for Speed

VI is a commonly used dynamic programming algorithm based on the Bellman equation for finding optimal policies of MDPs. Typically, VI requires repeatedly iterating over the state space of an MDP until the value function converges, at which point an optimal policy has been found. The version of VI presented in Algorithm 9.18 is a slightly modified version that was developed specifically for Lagrangian MDPs of aaCMDPs with a single constraint. It exploits the properties of aaCMDPs to find the optimal policy in a single loop over the state space, thereby arriving at a solution

Algorithm 9.18 Acyclic Absorbing Value Iteration

```
Input: $P$, $R^\lambda$, $R$, $D$, $S$
Output: $r^\lambda$, $r$, $d$, $\pi$
1: for all $s \in S$ in reverse topological order do
2:     $r^\lambda(s) \leftarrow 0$, $r(s) \leftarrow 0$, $d(s) \leftarrow 0$
3:     $r \leftarrow R^\lambda(s)$
4:     $\pi(s) \leftarrow \arg\max_a \sum_{s',r^\lambda} p(s', r^\lambda|s, a)[r^\lambda + R^\lambda(s')]$
5:     $r^\lambda(s) \leftarrow \sum_{s',r^\lambda} p(s', r^\lambda|s, \pi(s))[r^\lambda + R^\lambda(s')]$
6:     $r(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + R(s')]$
7:     $d(s) \leftarrow \sum_{s',d} p(s', d|s, \pi(s))[d + D(s')]$
8:     $\delta \leftarrow \max(\delta, |r - R^\lambda(s)|)$
9: return $r^\lambda$, $r$, $d$, $\pi$
```

much quicker than normal VI. It also evaluates the reward function $R(\pi)$ and cost function $D(\pi)$ at the same time.

There are two contributions here to consider. First, not only is the Lagrangian reward function evaluated for every state (line 5), but so are the reward and cost functions of the original CMDP (lines 6 and 7). This allows for an evaluation of the output policy π on these two functions by referencing their values at start state where $\beta(s) = 1$. Second, the for-loop only needs to commence once for all states as long as the proper ordering of states is used (line 1). This fact is established by the following theorem:

Theorem 9.2.2. *Given an MDP where no state can appear twice in a realization and with a single absorbing state that is reached with probability* 1*, value iteration can converge to the optimum value function in a single iteration.*

Proof. First, start by initializing the value of the absorbing state $R^\lambda(s_l) = 0$ and defining $\hat{S}$ as the set of all states that have been evaluated. Since the absorbing state has no action to any other state (other than itself, with transition cost $D(s_l) = 0$), $R^\lambda(s') = R^\lambda(s_l) = 0$ meaning the value of the absorbing state will never change, regardless of the number of iterations performed. Next, add s_l to $\hat{S}$ and evaluate some new state $s_i \in S \backslash \hat{S}$ where all possible actions at s_i lead only to states within $\hat{S}$. Such a state must exist, since it is known that all possible sequences of states end with the absorbing state (due to the probability of reaching the absorbing state being 1), and the lack of any loops allowing states to have actions that lead to states previously visited. s_i is evaluated according to the Bellman equation in line 5, and because s_i can only transition to s_l (the only state in $\hat{S}$ at this point) its value will also remain static over all iterations. Then, s_i is added to $\hat{S}$ and the process is repeated, resulting in a static value for every state that is added to $\hat{S}$. Eventually, the only state remaining is the starting state for the acyclic absorbing MDP, which is necessarily evaluated with an unchanging value. Thus, every state in S is eventually evaluated and determined to have a value that will not change across iterations of the value iteration algorithm. $\square$

Theorem 9.2.2 shows that, because of the acyclic and absorbing nature of the state space, every state needs to have its value evaluated only once, as long as they are done in the correct order. This means that the for-loop in Algorithm 9.18 (line 5) should choose $s \in S$ starting at the absorbing state s_l and work its way backwards as described in the proof, i.e. in reverse topological ordering. If done this way, there is no need to continually update the value function for every state until convergence, because convergence will have been achieved immediately. A valid question to ask is how to quickly sort the state space in reverse topological ordering. In cases where the state space has a temporal component, time can be used as long as all possible state transitions guarantee the progression of time. For the SOPCC, ordering of vertices in the initial path can be used, since every action guarantees moving to a vertex further along the path. In this case, Algorithm 9.18 runs in $\mathcal{O}(S^2)$ time, as the states can be presorted in $\mathcal{O}(S \log S)$.

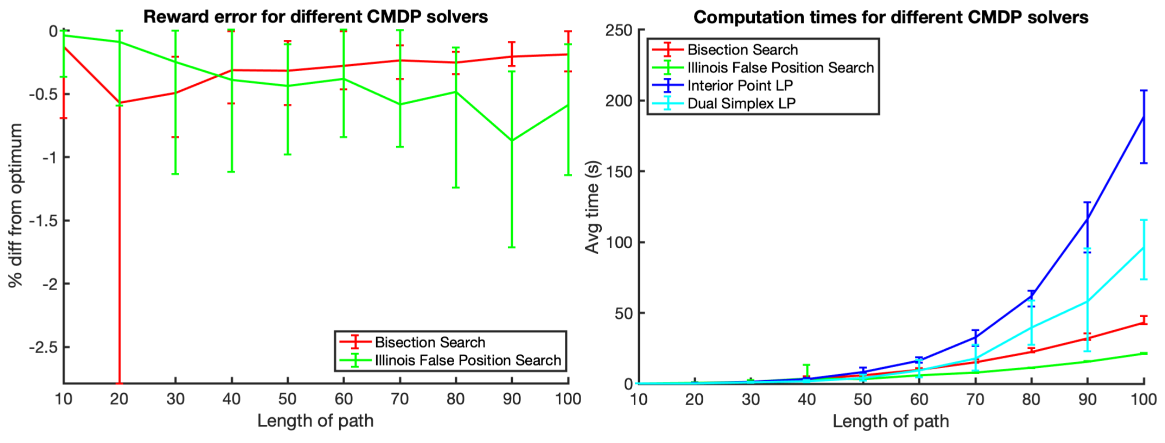
9.3 Comparing Lagrange to Linear Programming

To asses the efficiency of using the Lagrangian method when finding policies for aaCMDPs, the method as described in this chapter was tested against linear programming on a number of different aaCMDPs built for the SOPCC. Both the non-adaptive method from Chapter 6 and the adaptive path method from Chapter 7 using path trees were used. These were built for the SOPCC where positions for vertices $v \in V$ in a graph $G(V, E)$ are obtained by sampling the unit square with a uniform distribution and the reward for each $r(v)$ is a random sample from a uniform distribution on the interval $[0, 1]$. Edge costs are calculated the same as in section 6.4,

$$C(v_i, v_j) = \alpha d_{i,j} + \mathcal{E}\left(\frac{1}{(1-\alpha)d_{i,j}}\right) \tag{9.7}$$

where $d_{i,j}$ is the Euclidean distance between two vertices v_i, v_j and $0 < \alpha < 1$ which relates to the variance $((1-\alpha)d_{i,j})^2$ of the exponential distribution. As before, the initial path $\mathcal{P}$ is calculated by reducing the SOPCC to a deterministic OP using expected edge costs and solving using the S-Algorithm from [126]. The length of the initial path $|\mathcal{P}|$ was varied as this is the main factor effecting the time to find a solution. To obtain consistent length paths, the budget B was allowed to vary and the S-Algorithm was run multiple times for each trial until a suitable budget and length of path was found. These extra runs were not included as part of the results. Other parameters were fixed, with $\alpha = 0.5$, failure constraint $P_f = 0.05$, max reward deviation $\varepsilon = 0.1$, number of time steps $|\mathbb{T}| = 10$, and $\theta = 0.0001$. For each data point, 10 random graphs were created and a CMDP was constructed for the SOPCC using a budget resulting in the appropriate length of initial path.

Each CMDP was solved using the Lagrangian method as described in section 9.2 with both Bisection Search and Illinois False Position Search as the means of finding the appropriate Lagrange multipliers. These results were compared against two standard linear programming solution methods, the Interior Point algorithm and the



(a) (b)

Figure 9.1: Results of the non-adaptive SOPCC CMDP solvers on lengths of paths up to 100 vertices. Shown are averages as well as minimum and maximums. (a) The error of expected reward collected for the Lagrangian methods of solving the CMDP for each SOPCC. (b) The required time to solution using different methods of solving the CMDP for each SOPCC.

Dual Simplex Algorithm. All coding was done in Matlab and the built-in functions for linear programming were used. There exists more efficient linear programming solvers, however implementing everything in Matlab allows for establishing a consistent standard to evaluate the scalability of the various methods. Each of the tests shown were performed on a Linux computer running an Intel Core i7 6700k processor with 32GB of ram.

To begin, the non-adaptive SOPCC algorithm was used to build CMDPs. Figure 9.1 shows the results of the tests on lengths of initial paths between 10 and 100 vertices. The chart in Figure 9.1b shows the average, minimum, and maximum computation times for each of the 10 randomized graphs across all lengths of paths. Initially, there is little noticeable difference between each of the 4 methods, however when $|\mathcal{P}| \geq 50$, the trends start to become apparent. The computation times for Interior Point method and the Dual Simplex method both start to grow very quickly in relation to $|\mathcal{P}|$, whereas the Lagrangian approaches using Bisection Search and Illinois False Position Search do not take off as quickly. The linear programming methods also develop large spreads in computation time, whereas the Lagrangian methods stay very consistent. For a path of 100 vertices, the Lagrangian using Illinois False Position search is the clear winner in terms of speed, with an average time of $21.3s$ to solution, followed by Lagrangian with Bisection search with $43.1s$, then the Dual Simplex linear programming method with $96.3s$ and lastly the Interior Point linear programming method with $188.7s$.

The chart in Figure 9.1a shows the percentage difference from optimal either of the two Lagrangian approaches are in terms of expected collected reward. The average as well as the minimum and maximum differences are displayed. Since $\varepsilon = 0.1$ for

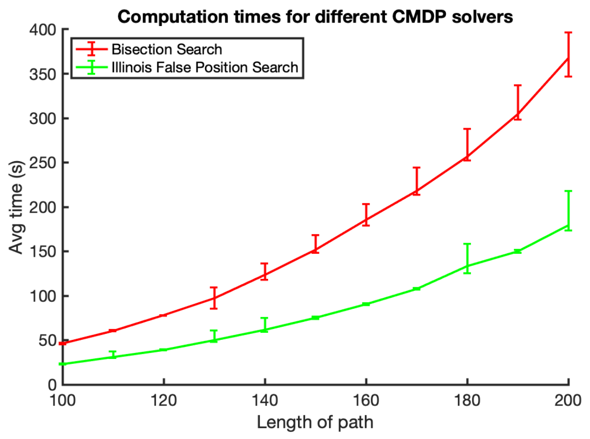
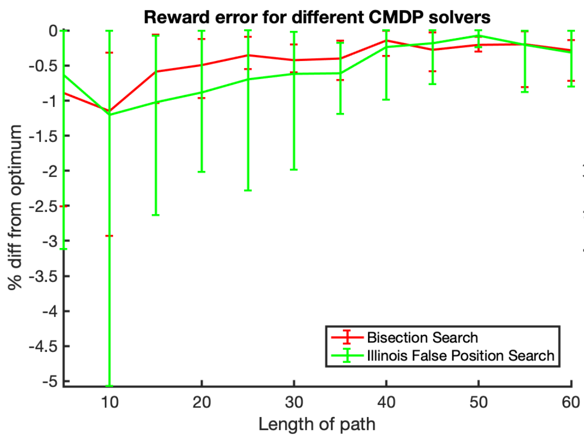


Figure 9.2: The required time to solution using the two Lagrangian methods of solving the CMDP for each SOPCC, on lengths of paths from 100 to 200 vertices.

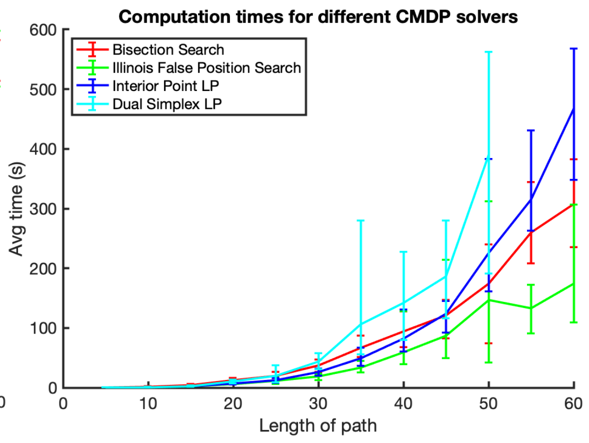
all of the tests, the maximum percentage difference that either search will achieve is 10%, however in practice the actual difference is much smaller than the bound. The largest error on reward for the Bisection search was only 2.79%, found when $|\mathcal{P}| = 20$, and every other trial had errors less than 1%. When $\mathcal{P}$ was 10, 20, and 40 vertices in length, some trials resulted in errors of 0%, showing that it is indeed possible for the Lagrangian method to optimally solve a CMDP. The largest percentage difference in reward from optimal for the Illinois False Position search was 1.71% when $|\mathcal{P}| = 90$, and some trials achieved zero error when the length of the path was 10, 20, 30, 40, and 60 vertices.

The Lagrangian methods were also run on SOPCC CMDPs built using with initial paths from 100 vertices in length up to 200 vertices. The results are shown in Figure 9.2. As expected, the trends continue for both Lagrangian methods, with the Bisection search approach taking roughly double the amount of time to solution as the Illinois False Position approach. Due to time and memory constraints, the linear programming methods were impractical to use and therefore not included as part of these tests.

Finally, the adaptive path method for the SOPCC was used to compare the Lagrangian methods to the linear programming methods with the resulting CMDPs, with the results shown in Figure 9.3. Though a path tree creates a more complex state space, the resulting CMDP is still acyclic and absorbing and therefore the Lagrangian approach exploiting properties of this type of state space still applies. As before, the length of initial path was varied as a proxy to the size of the resulting CMDP. Despite the adaptive path method utilizing the branch heuristics from section 7.3 with $k_b = 5$, the time to solution for all methods still grows much faster than before due to the larger state and action spaces. Compared to the linear programming methods, the Lagrangian approaches are still favorable in terms of the amount of time required to compute a solution when $|\mathcal{P}| \geq 100$. The same trends as seen on the non-adaptive SOPCC CMDPs are seen here, with the Illinois False Position Search



(a)



(b)

Figure 9.3: Results of the adaptive SOPCC CMDP solvers on lengths of paths up to 120 vertices. Shown are averages as well as minimum and maximums. (a) The error of expected reward collected for the Lagrangian methods of solving the CMDP for each SOPCC. (b) The required time to solution using different methods of solving the CMDP for each SOPCC.

taking on average $147s$ versus the Dual Simplex Linear program taking on average $391s$ for an initial path length of 100 vertices. When computing results for initial paths longer than 100 vertices, the Dual Simplex solver took too long to complete and computation was suspended before a policy was obtained. In terms of reward error, the Lagrangian methods seem to have performed slightly worse than before, however they are still well within the given allowed error of 10%. The worst cases came from when $|\mathcal{P}| = 20$, with the Illinois False Position search method achieving a 5.08% difference from the optimum reward and the Bisection search method managing a 2.93% difference. Both Lagrangian search methods were able to attain 0% errors on multiple occasions for a few different budgets. These results show the presented Lagrangian methods are very quick at solving large aaCMDPs, and can do so without losing much optimality.

9.4 Conclusion

The methods presented to solve the SOPCC in Chapter 6 and Chapter 7 require finding policies to very large CMDPs, especially when the length of the initial path is long. This is problematic because conventional CMDP solvers use linear programming to find optimal policies, and the time to solution grows very quickly as the size of the state and action spaces increases. One way of calculating policies quicker is to use the Lagrangian method of solving CMDPs, which allows for the use of dynamic programming methods typically only available to MDPs. By exploiting the design of the specially constructed state space for the SOPCC, it is possible to quickly produce policies for the Lagrangian MDPs. Then, it is possible to adjust these policies to find the appropriate Lagrangian multiplier by using a bracketed root finding method, which also guarantees that the resulting policy will be within a bounded distance from the optimal. These facts were shown to be true when tested on randomizes instances of the SOPCC of various sizes.

Chapter 10

Final Thoughts

10.1 Conclusions

Throughout this dissertation, routing algorithms for autonomous agents operating in aisle-like environments are discussed. Chapter 1 provided the motivation, which involves the use of robots in vineyards to precisely adjust irrigation emitters on a vine-by-vine basis in order to increase water use efficiency. There are challenges to navigating a robot within a vineyard to provide optimal adjustment. These challenges are highly specific to the environmental model used, which was framed as an Aisle Graph (AG), and require solving computationally difficult problems in order to obtain effective solutions. The problems discussed throughout are the Orienteering Problem (OP), the Team Orienteering Problem (TOP), the Bi-Objective Orienteering Problem (BOOP), and the Stochastic Orienteering Problem with Chance Constraints (SOPCC), and an overview for each was given in Chapter 2.

Chapter 3 discusses how to create AGs, the vertex-edge graph model which underpins all the following presented algorithms, and also the heuristic methods solving the Aisle Graph Orienteering Problem (AGOP). AGs belong to a special class of graphs called Bipartite Planar graphs of degree 3 ($\mathcal{BP}3$), and express an aisle-like structure which gives AGs their name. Greedy Partial Row (GPR), one of the two AGOP heuristics outlined in the chapter, was designed specifically to make use of the $\mathcal{BP}3$ structure in order to quickly build efficient paths for agents operating on AGs. It was shown to be faster than general case orienteering heuristics while also also giving higher quality solutions, verified by simulating irrigation emitter adjustment problems for vineyards.

Chapter 4 discusses how to utilize the knowledge learned from single agent orienteering in vineyards to solve the Aisle Graph Team Orienteering Problem (AGTOP). A big challenge with the AGTOP is the coordination of multiple agents across the graph, which requires that none of the agents cross each other's path inside of a row. This is a problem associated with a team of robots operating in a vineyard, where each row is narrow enough to limit only one robot within it at a time without risk for collision. Three heuristics based on GPR are given which are able to overcome this challenge, coordinating each agent in space-time while also efficiently scaling to large graphs with large numbers of agents. Again, these were shown to be more effective than general case heuristics and much faster for the same size team and problem.

The Aisle Graph Bi-Objective Orienteering Problem (AGBOOP) was discussed in Chapter 5. The main purpose of a bi-objective approach to routing in vineyards is the ability for a robot to carry multiple payloads and perform different tasks. One in particular is soil moisture sampling, which can be done along side of irrigation adjustment. Two types of AGBOOPs were discussed, the Dual Maximization (AGDMBOOP) variant, which aims to maximize for both objectives, and the Objective Constraint (AGOCBOOP) variant, which aims to constraint one of the objectives to a minimum value while maximizing the other. By extending GPR to optimize paths for multiple reward functions, it was shown that an agent navigating across an AG can balance two objectives at once for either of the two variations of the AGBOOP. Simulations on vineyard data proved that the extended heuristics are suitable for real-world problems where irrigation adjustment and soil moisture sampling are necessary tasks.

One important aspect of real world robotics problems that is often overlooked is the stochasticity of robotic movement. In particular, field conditions may compromise the assumption of determinism regarding mobility. For a ground robot, this can manifest as needing unexpected amounts of time when moving from place to place. Chapter 6 attempts to address this issue for the OP, by studying a problem called the Stochastic Orienteering Problem with Chance Constraints (SOPCC). The SOPCC rationalizes stochastic movements across edges in a graph, and reasons about choices that an agent can make when deciding where to go next. A policy is created which tells the agent how to maximize the expected reward of its path while also restricting its chance of overrunning the budget to a defined maximum probability. A heuristic method is given that utilizes any deterministic OP solver to create an initial path and creates a policy that tells an agent where and when it should take shortcuts along the path to reach the goal vertex on time. The method is validated on randomized generic graphs to show its suitability but lacks adaptability and scalability.

Chapter 7 discusses two ways to modify the SOPCC method into an adaptive algorithm. The first provides an adaptive way to choose the time intervals to create an appropriate state space. It relies on simulating the initial path over several trials and finding the distribution of arrival times for each vertex. The arrival times are used in combination with a uniform discretization of the time dimension to create a new adaptive time discretization for every vertex. The second provides an adaptive path over which an agent can move. A path tree is created by finding states which are likely to be used but provide less than ideal utility and computing an additional branch that redirects the agent to collect more reward. Combining both of these method together creates an adaptive heuristic for the SOPCC which is more productive in terms of reward collection than the non-adaptive SOPCC method.

To correct the lack of scalability for the SOPCC, Chapter 8 presents a mechanism by which vertices in a path can be aggregated into a set of compound vertices which are each representative of multiple consecutive vertices. While it reduces the number of available shortcuts, it also greatly reduces the amount of computation time required to find a policy that satisfies the chance constraint. A way to apply aggregation to orienteering paths created by GPR was also examined, which utilized full-rows and partial-rows as natural aggregation positions, thus allowing both the non-adaptive and adaptive SOPCC methods to work on AGs with many vertices. Once again vineyard data was used to simulate irrigation adjustment problems where a robot needs to navigate with randomized edge costs, and results confirm that the provided aggregation method for the SOPCC is able to scale to large problem sizes.

While scalability of the SOPCC has been addressed using agregation, it is a compromise that compresses the state space to a manageable size rather than solving a larger problem quicker. Chapter 9 attempts to correct this by using the Lagrangian method to replace linear programming when finding policies for the SOPCC. The SOPCC CMDP is reduced to a simpler Lagrangian MDP over which a much faster dynamic programming solver can find a policy. By using a bracketed root finding method such as Binary search or Illinois False Position search, the optimal Lagrangian multiplier can be approximated with two nearby values that satisfy an error bound on the reward. The policies for these two Lagrange multiplier values can be mixed together to provide a policy for the CMDP which keeps the constraint active and is close to optimal. Experimentation on randomized graphs show that the Lagrangian method of solving CMDPs scales better than the linear programming method and can produce SOPCC policies for larger problems.

10.2 Future Work

There is more work to be done regarding planning algorithms for robots operating in vineyards to optimize water usage. The algorithms presented are only useful with prior knowledge given about a vineyard's needed irrigation adjustment, and they also make some assumptions that may not be true in practice. Future work should focus on challenging these assumptions to make the irrigation adjustment more robust and engineering new algorithmic approaches that achieve better results than obtained thus far.

10.2.1 Non-uniform Costs and Different Vineyard Shapes

One big assumption made in this dissertation is that an AG is always rectangular in shape, and uniform in cost. Real-life vineyard blocks, however, do not fit these assumptions. A block will most likely be missing more than a few vines, due to removal for various reasons, and therefore some vertexes will be missing in its graph representation. Not all vineyards are rectangular in shape either, some may be triangular or irregularly shaped due to natural geography, human activity, or land usage requirements. It is often the case that they exist on plots of land that are not flat ground, giving rise to non-uniform costs between vertexes and possibly different costs depending on the direction of travel. Finally, it is even possible that vineyards have no grid infrastructure such as trellises limiting movement, which is sometimes the case with older vineyards, and the restrictions imposed by this structure are no longer a concern. In all of these cases, the AGOP and AGTOP solvers will need to be rethought, or perhaps even dismissed for better alternatives, to deal with these less orderly circumstances.

10.2.2 Improving GPR

One disadvantage to the GPR heuristic used throughout is that it works using a greedy paradigm. Greedy algorithms are locally optimal, meaning they make the best choice for the next decision, maximizing the outcome immediately. However, a series of greedy choices does not always attain the best overall outcome, and this is especially true for NP-hard problems like the OP. The optimal combination of choices is non-obvious because every previous decision effects the results of future decisions. This is evident in all of the aforementioned solvers by looking at the resulting paths. A path may cross two rows that are right next to each other but they may not be directly connected, instead many other full-rows or partial-rows can be traversed between them. This is not optimal, as budget will be wasted traveling to those other rows and back. Effort should be exerted to produce a path that wastes a minimal amount of budget.

One method to solve this problem is to use a k-Horizon paradigm instead of a strictly greedy one. k-Horizon algorithms are a form of receding, moving, or rolling horizon algorithms that look at sequences of k decisions before making a choice, instead of a single decision as with a greedy algorithm. By looking k decisions ahead, it is possible to find a choice that more globally optimal than a simple greedy heuristic. The disadvantage of this type of algorithm is the computational complexity, as it increases the number of iterations required by k. An infinite horizon algorithm would perform the best, as it will look at every possible combination of future decisions, making only the optimal choice, however this is no different than using a tree search algorithm. By redesigning the GPR algorithm to use a k-Horizon, it will be possible to produce better results that waste less budget.

Another way to conserve budget when building paths on AGs for the OP and its derivatives is to do it after running the heuristics instead of during. The sequence of visitations for each full-row and partial-row in the path can be reordered linearly to cut excess travel costs. This idea is the result of the observation that paths will often waste budget running up and down the sides of the AG, passing rows multiple times before eventually including them in the path. Once rearrangements are made, newly freed budget can then be used by resetting the feasibility of unvisited vertexes and continuing where the heuristic algorithm left off. This process can be repeated until no change happens on the last run, eventually reaching an equilibrium that gives a much more optimal result than initially. This approach has an advantage over the k-Horizon method in that it will not multiply the runtime by k. However, it also has the disadvantage that the runtime will be dependent on the newly freed budget, which is impossible to know ahead of time.

10.2.3 Non-homogeneous Agents

Another assumption made by the algorithms solving the AGTOP is that all of the agents are exactly the same. That is, they all have the same reward and cost functions over the given AG. It is possible that the agents cooperating together are non-homogeneous; there could be multiple types of robots that move at different speeds and there could be humans that perform the same tasks but with different levels of efficiency. This would lead to different reward and cost functions that are agent dependent, resulting in a need to plan and coordinate all agents according to their ability. Allowing non-homogeneous agents to work together requires some changes to how the AGTOP algorithms work. Calculations for the heuristic values need to account for the different rewards and costs associated with each agent, and calculations for the time conflicts need to account for the differences in movement speed as well. It may be beneficial to change the operations of SGPR and PGPR to iterate loops on a time basis rather than on full/partial-row and feasible vertex basis. This would fundamentally modify how the algorithms work, however, changing their complexities and may possibly make them less computationally efficient. That is just one method of reckoning dissimilar agents solving the AGTOP and there may be others.

10.2.4 Online Path Planning

When navigating a vineyard to adjust irrigation, soil moisture sampling can also be carried out as a supplemental task. This leads to a question of whether it is possible and practical to use new data collected from the sampling to modify prior knowledge of the vineyards moisture status and change the irrigation based on the new knowledge. This would require online updating of the robot's path, however it is not a trivial task. For example, in order to use GPR for online path planning, the algorithm will need to be changed to allow starting a path from any vertex in the graph, so as to allow the agent to start from its current location after the reward function is updated. Additionally, there will be a need for some method to amend the reward function based on the new data, and this will not be straightforward because of how soil drainage and water percolation can effect the moisture of surrounding soil when one irrigation emitter is changed. Thus, incorporating additional information to an AG, whether it changes old data or provides new knowledge, modifies the way the OP and TOP must be solved for vineyards.

Bibliography

[1] California agricultural exports 2019-2020. Technical report, California Department of Food and Agriculture, 2020.

[2] Andr Silva Aguiar, Filipe Neves dos Santos, Jos Boaventura Cunha, Hber Sobreira, and Armando Jorge Sousa. Localization and mapping for robots in agriculture and forestry: A survey. *MDPI Robotics*, 9(4):97:1–97:23, 2020.

[3] Bilal Hazim Younus Alsalam, Kye Morton, Duncan Campbell, and Felipe Gonzalez. Autonomous uav with vision based on-board decision making for remote sensing and precision agriculture. In *IEEE Aerospace Conference*, pages 1–11, 2017.

[4] Eitan Altman. *Constrained Markov Decision Processes - Stochastic Modeling, Vol. 7.* Chapman and Hall/CRC, 1999.

[5] Eitan Altman and Flos Spieksma. The linear program approach in multi-chain markov decision processes revisited. *Mathematical Methods of Operations Research*, 42(2):169–188, 1995.

[6] Rajkishan Arikapudi and Stavros G. Vougioukas. Robotic tree-fruit harvesting with telescoping arms: A study of linear fruit reachability under geometric constraints. *IEEE Access*, 9:17114–17126, 2021.

[7] Baruch Awerbuch, Yossi Azar, Avrim Blum, and Santosh Vempala. Improved approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 277–283, 1995.

[8] Benjamin J. Ayton and Brian C. Williams. Vulcan: A monte carlo algorithm for large chance constrained mdps with risk bounding functions. arXiv, 2018.

[9] Jacopo Banfi, Nicola Basilico, and Francesco Amigoni. Intractability of time-optimal multirobot path planning on 2d grid graphs with holes. *IEEE Robotics and Automation Letters*, 2(4):1941–1947, 2017.

[10] Nikhil Bansal, Avrim Blum, Shuchi Chawla, and Adam Meyerson. Approximation algorithms for deadline-tsp and vehicle routing with time-windows. In *Proceedings of the Thirty-Sixth Anuual ACM Symposium on Theory of Computing*, pages 166–174, 2004.

[11] Nikhil Bansal and Viswanath Nagarajan. On the adaptivity gap of stochastic orienteering. *Mathematical Programming*, 154:145–172, 2015.

[12] Suchet Bargoti and James P. Underwood. Image segmentation for fruit detection and yield estimation in apple orchards. *Journal of Field Robotics*, 34(6):1039–1060, 2017.

[13] Antonio Barrientos, Julian Colorado, Jaime del Cerro, Alexander Martinez, Claudio Rossi, David Sanz, and Joo Valente. Aerial remote sensing in agriculture: A practical approach to aera coverage and path planning for fleets of mini aerial robots. *Journal of Field Robotics*, 28(5):677–689, 2011.

[14] Christopher Bayliss, Angel A. Juan, Christine S.M. Currie, and Javier Panadero. A learnheuristic approach for the team orienteering problem with aerial drone motion constraints. *Applied Soft Computing Journal*, 92:106280:1–106280:19, 2020.

[15] Hiba Bederina and Mhand Hifi. A hybrid multi-objective evolutionary algorithm for the team orienteering problem. In *International Conference on Control, Decision and Information Technologies*, pages 898–903, 2017.

[16] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[17] Ron Berenstein and Yael Edan. Human-robot collaborative site-specific sprayer. *Journal of Field Robotics*, 34(8):1519–1530, 2017.

[18] Ron Berenstein, Ohad Ben Shahar, Amir Shapiro, and Yael Edan. Grape clusters and foliage detection algorithms for autonomous selective vineyard sprayer. *Intelligent Service Robotics*, 3(4):233–243, 2010.

[19] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. 1 and 2*. Athena Scientific, 1995.

[20] Dimitri P. Bertsekas. *Reinforcement Learning and Optimal Control*, chapter Aggregation, pages 308–340. Athena Scientific, 2019.

[21] Frederick J. Beutler and Keith W. Ross. Optimal policies for controlled markov chains with a constraint. *Journal of Mathematical Analysis and Applications*, 112(1):236–252, 1985.

[22] Avrim Blum, Shuchi Chawla, David R. Karger, Terran Lane, Adam Meyerson, and Maria Minkoff. Approximation algorithms for orienteering and discounted-reward tsp. *SIAM Journal on Computing*, 37(2):653–670, 2007.

[23] D. Bochtis, H.W. Griepentrog, S. Vougioukas, P. Busato, R. Berruto, and K. Zhou. Route planning for orchard operations. *Computers and Electronics in Agriculture*, 113:51–60, 2015.

[24] Vivek Borkar and Rahul Jain. Risk-constrained markov decision processes. *IEEE Transactions on Automatic Control*, 59(9):2574–2579, 2014.

[25] Jakob Bossek, Christian Grimme, Stephan Meisel, Gnter Rudolph, and Heike Trautmann. Bi-objective orienteering: Towards a dynamic multi-objective evolutionary algorithm. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 516–528, 2019.

[26] Jakob Bossek, Christian Grimme, and Heike Trautmann. Dynamic bi-objective routing of multiple vehicles. In *Genetic and Evolutionary Computation Conference*, pages 166–174, 2020.

[27] Tom Botterill, Scott Paulin, Richard Green, Samuel Williams, Jessica Lin, Valerie Saxton, Steven Mills, XiaoQi Chen, and Sam Corbett-Davies. A robot system for pruning grape vines. *Journal of Field Robotics*, 34(6):1100–1122, 2017.

[28] Sylvain Boussier, Dominique Feillet, and Michel Gendreau. An exact algorithm for team orienteering problems. *4OR*, 5(3):211–230, 2007.

[29] Steven E. Butt and Tom M. Cavalier. A heuristic for the multiple tour maximum collection problem. *Computers and Operations Research*, 21(1):101–111, 1994.

[30] Steven E. Butt and David M. Ryan. An optimal solution procedure for the multiple tour maximum collection problem using column generation. *Computers and Operations Research*, 26(4):427–441, 1999.

[31] Ann M. Campbell, Michel Gendreau, and Barrett W. Thomas. The orienteering problem with stochastic travel and service times. *Annals of Operations Research*, 186(1):61–81, 2011.

[32] Stefano Carpin, Derek Burch, Nicola Basilico, Timothy H. Chung, and Mathias Kölsch. Variable resolution search with quadrotors: Theory and practice. *Journal of Field Robotics*, 30(5):685–701, 2013.

[33] Stefano Carpin, Marco Pavone, and Brian M. Sadler. Rapid multirobot deployment with time constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1147–1154, 2014.

[34] I-Ming Chao, Bruce L. Golden, and Edward A. Wasil. The team orienteering problem. *European Journal of Operational Research*, 88:464–474, 1996.

[35] M. M. Chaves, T. P. Santos, C. R. Souza, M. F. Ortuo, M. L. Rodrigues, C. M. Lopes, J. P. Maroco, and J. S. Pereira. Deficit irrigation in grapevine improves water-use efficiency while controlling vigor and production quality. *Annals of Applied Biology*, 150, 2007.

[36] Chandra Chekuri, Nitish Korula, and Martin Pl. Improved algorithms for orienteering and related problems. *ACM Transactions on Algorithms*, 8(3):23:1–23:27, 2012.

[37] Ke Chen and Sariel Har-Peled. The orienteering problem in the plane revisited. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 1–12, 2006.

[38] Yu-Han Chen, Wei-Ju Sun, and Tsung-Che Chiang. Multiobjective orienteering problem with time windows: An ant colony optimization algorithm. In *Conference on Technologies and Applications of Artificial Intelligence*, pages 128–135, 2015.

[39] Xavier Chon, Cornelis van Leeuwen, Denis Dubourdieu, and Jean Pierre Gaudillere. Stem water potential is a sensitive indicator of grapevine water status. *Annals of Botany*, 87:477–483, 2001.

[40] Yin-Lam Chow, Marco Pavone, Brian M. Sadler, and Stefano Carpin. Trading safety versus performance: Rapid deployment of robotic swarms with robust performance constraints. *Journal of Dynamic Systems, Measurement, and Control*, 137:031005.1–031005.11, 2015.

[41] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[42] Cyrus Derman and Morton Klein. Some remarks on finite horizion markovian decision models. *Operations Research*, 13(2):169–340, 1965.

[43] Martin Desrochers and Gilbert Laporte. Improvements and extensions to the miller-tucker-zemlin subtour elimination constraints. *Operations Research Letters*, 10:27–36, 1991.

[44] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[45] Irina Dolinskaya, Zhenyu Shi, and Karen Smilowitz. Adaptive orienteering problem with stochastic travel times. *Transportation Research Part E*, 109:1–19, 2018.

[46] M. Dowell and P. Jarratt. A modified regula falsi method for computing the root of an equation. *BIT Numerical Mathematics*, 11:168–174, 1971.

[47] Racha El-Hajj, Duc-Cuong Dang, and Aziz Moukrim. Solving the team orienteering problem with cutting planes. *Computers and Operations Research*, 74:21–30, 2016.

[48] Alejandro Estrada-Moreno, Albert Ferrer, Angel A. Juan, Javier Panadero, and Adil Bagirov. The non-smooth and bi-objectivve team orienteering problem with soft constraints. *Mathematics*, 8(9):1461, 2020.

[49] Lanah Evers, Kristiaan Glorie, Suzanne van der Ster, Ana Isabel Barros, and Herman Monsuur. A two-stage approach to the orienteering problem with stochastic weights. *Computers and Operations Research*, 43:248–260, 2014.

[50] Seyedshams Feyzabadi and Stefano Carpin. Multi-objective planning with multiple high level task specifications. In *IEEE International Conference on Robotics and Automation*, pages 5483–5490, 2016.

[51] Seyedshams Feyzabadi and Stefano Carpin. Planning using hierarchical constrained markov decision processes. *Autonomous Robots*, 41:1589–1607, 2017.

[52] Carlo Filippi and Elisa Stevanato. Approximation schemes for bi-objective combinatorial optimization and their application to the tsp with profits. *Computers and Operations Research*, 40(10):2418–2428, 2013.

[53] Matteo Fischetti, Juan Jos Salazar Gonzlez, and Paolo Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10(2), 1998.

[54] Spyros Fountas, Nikos Mylonas, Ioannis Malounas, Efthymios Rodias, Christoph Hellman Santos, and Erik Pekkeriet. Agricultural robotics for field operations. *MDPI Sensors*, 20:2672, 2020.

[55] B. L. Fox and D. M. Landi. Searching for the multiplier in one-constraint optimization problems. *Operations Research*, 18(2):193–373, 1970.

[56] Naveen Garg. Saving an epsilon: A 2-approximation for the k-mst problem in graphs. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 396–402, 2005.

[57] David V. Gealy, Stephen McKinley, Menglong Gou, Lauren Miller, Stavros Vougioukas, Joshua Viers, Stefano Carpin, and Ken Goldberg. Co-robotic device for automated tuning of emitters to enable precision irrigation. In *IEEE Conference on Automation Science and Engineering*, pages 922–927, 2016.

[58] Edward P. Glenn, Christopher M. U. Neale, Doug J. Hunsaker, and Pamela L. Nagler. Vegetation index-based crop coefficients to estimate evapotranspiration by remote sensing in agricultural and natural ecosystems. *Hydrological Processes*, 25:4050–4062, 2011.

[59] Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics*, 34:307–318, 1987.

[60] Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. Orienteering problem: A survey of recent variants, solution approaches, and applications. *European Journal of Operational Research*, 255(2):315–332, 2016.

[61] Anupam Gupta, Ravishankar Krishnaswamy, Viswanath Nagarajan, and R. Ravi. Approximation algorithms for stochastic orienteering. *ACM-SIAM Symposium on Discrete Algorithms*, pages 1522–1538, 2012.

[62] Anupam Gupta, Ravishankar Krishnaswamy, Viswanath Nagarajan, and R. Ravi. Running errands in time: Approximation algorithms for stochastic orienteering. *Mathematics of Operations Research*, 40(1):56–79, 2014.

[63] Michael Halstead, Christopher McCool, Simon Denman, Tristan Perez, and Clinton Fookes. Fruit quantity and ripeness estimation using a robotic vision system. *IEEE Robotics and Automation Letters*, 3(4):2995–3002, 2018.

[64] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[65] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The Technology Press of M.I.T., Cambridge, MA and John Wiley and Sons Inc., New York, NY, 1960.

[66] Wanzhe Hu, Mahdi Fathi, and Panos M. Pardalos. A multi-objective evolutionary algorithm based on decomposition and constraint programming for the multi-objective team orienteering problem with time windows. *Applied Soft Computing Journal*, 73:383–393, 2018.

[67] Xin Huang, Ashkan Jasour, Matthew Deyo, Andreas Hofmann, and Brian C. Williams. Hybrid risk-aware conditional planning with applications in autonomous vehicles. In *IEEE Conference on Decision and Control*, pages 3608–3614, 2018.

[68] E. Raymond Hunt and Craig S. T. Daughtry. What good are unamnned aircraft systems for agricultural remote sensing and precision agriculture? *International Journal of Remote Sensing*, 39(15-16):5345–5376, 2018.

[69] D. S. Intrigliolo, C. Ballester, and J. R. Castel. Carry-over effects of deficit irrigation applied over seven seasons in a developing japanese plum orchard. *Agricultural Water Management*, 128:13–18, 2013.

[70] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, November 1982.

[71] Stefan Jorgensen, Robert H. Chen, Mark B. Milam, and Marco Pavone. The team surviving orienteers problem: Routing robots in uncertain environments with survival constraints. In *International Conference on Robotic Computing*, pages 227–234, 2017.

[72] Xinyue Kan, Thomas C. Thayer, Stefano Carpin, and Konstantinos Karydis. Task planning on stochastic aisle graphs for precision agriculture. *IEEE Robotics and Automation Letters*, 6(2):3287–3294, 2021.

[73] Hajime Kawai and Naoki Katoh. Variance constrained markov decision process. *Journal of the Operations Research*, 30(1):88–100, 1987.

[74] Liangjun Ke, Claudia Archetti, and Zuren Feng. Ants can solve the team orienteering problem. *Computers and Industrial Engineering*, 54:648–665, 2008.

[75] James A. Kennedy, Mark A. Matthews, and Andrew L. Waterhouse. Effect of maturity and vine water status on grape skin and wine flavonoids. *American Journal of Enology and Viticulture*, 53(4):268–274, 2002.

[76] Morteza Keshtkaran, Koorush Ziarati, Andrea Bettinelli, and Daniele Vigo. Enhanced exact solution methods for the team orienteering problem. *International Journal of Production Research*, 54(2):591–601, 2016.

[77] Sami Khairy, Prasanna Balaprakash, Lin X. Cai, and Yu Cheng. Constrained deep reinforcement learning for energy sustainable multi-uav based random access iot networks with noma. *arXiv*, pages 1–14, 2020.

[78] Levente Kocsis and Csaba Szepesvri. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

[79] Gary J. Koehler. A case for relaxation methods in large scale linear programming. In *IFAC Symposium on Large Scale Systems Theory and Applications*, volume 9, pages 293–302, 1976.

[80] Vikram Krishnamurthy. *Partially Observed Markov Decision Processes.* Cambridge University Press, 2016.

[81] Pankaj Kumar and G. Ashok. Design and fabrication of smart seed sowing robot. In *International Conference on Advanced Materials and Modern Manufacturing*, pages 354–358, 2021.

[82] Polina Kurtser, Ola Ringdahl, Nati Rotstein, Ron Berenstein, and Yael Edan. In-field grape cluster size assessment for vine yield estimation using a mobile robot and a consumer level rgb-d camera. *IEEE Robotics and Automation Letters*, 5(2):2031–2038, 2020.

[83] Hoong Chuin Lau, William Yeoh, Pradeep Varakantham, Duc Thien Nguyen, and Huaxing Chen. Dynamic stochastic orienteering problems for risk-aware applications. In *Conference on Uncertainty in Artificial Intelligence*, pages 1–11, 2012.

[84] Jongmin Lee, Geon-Hyeong Kim, Pascal Poupart, and Kee-Eung Kim. Monte-carlo tree search for constrained mdps. In *ICML/IJCAI/AAMAS Workshop on Planning and Learning*, pages 1–9, 2018.

[85] Jongmin Lee, Geon-Hyeong Kim, Pascal Poupart, and Kee-Eung Kim. Monte-carlo tree search for constrained pomdps. In *Conference on Neural Information Processing Systems*, page 79347943, 2018.

[86] Zhengqi Li and Volkan Isler. Large scale image mosaic construction for agricultural applications. *IEEE Robotics and Automation Letters*, 1(1):295–302, 2016.

[87] Shih-Wei Lin. Solving the team orienteering problem using effective multi-start simulated annealing. *Applied Soft Computing*, 13:1064–1073, 2013.

[88] Stephen J. Maher, Tobias Fischer, Tristan Gally, Gerald Gamrath, Ambros Gleixner, Robert Lion Gottwald, Gregor Hendel, Thorsten Koch, Marco E. Lübbecke, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert Schwarz, Felipe Serrano, Yuji Shinano, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The scip optimization suite 4.0. Technical report, Optimization Online, 2017.

[89] Rodrigo Martn-Moreno and Miguel A. Vega-Rodrguez. Multi-objective artificial bee colony algorithm applied to the bi-objective orienteering problem. *Knowledge-Based Systems*, 154:93–101, 2018.

[90] Piotr Matl, Pamela C. Nolz, Ulrike Ritzinger, Mario Ruthmair, and Fabien Tricoire. Bi-objective orienteering for personal activity scheduling. *Computers and Operational Research*, 82:69–82, 2017.

[91] Yi Mei, Flora D. Salim, and Xiaodong Li. Efficient meta-heuristics for the multi-objective time-dependent orienteering problem. *European Journal of Operational Research*, 254:443–457, 2016.

[92] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, 7(4):326–329, 1960.

[93] Patrick Ngatchou, Anahita Zarei, and M. A. El-Sharkawi. Pareto multi objective optimization. In *International Conference on Intelligent Systems Application to Power Systems*, pages 84–91, 2005.

[94] M. A. Oliver and R. Webster. Kriging: A method of interpolation for geographical information systems. *International Journal of Geographical Information Systems*, 4(3):313–332, 1990.

[95] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[96] Christos H. Papadimitriou. The euclidean traveling salesman problem is np-complete. *Theoretical Computer Science*, 4, 1977.

[97] Lynne E. Parker. Path planning and motion coordination in multiple robot teams. In *Encyclopedia of Complexity and System Science*, pages 1–24. Springer, 2009.

[98] Sophie N. Parragh and Fabien Tricoire. Branch-and-bound for bi-objective integer programming. *INFORMS Journal on Computing*, 31(4):805822, 2019.

[99] David Pisinger and Stefan Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34(8):2403–2435, 2007.

[100] Santosh Pitla, Sreekala Bajwa, Santosh Bhusal, Tom Brumm, Tami M. Brown-Brandl, Dennis R. Buckmaster, Isabella Condotta, John Fulton, Todd J. Janzen, Manoj Karkee, Mario Lopez, Robert Moorhead, Michael Sama, Leon Schumacher, Scott Shearer, and Alex Thomasson. Ground and aerial robots for agricultural production: Opportunities and challenges. Technical Report 70, Council for Agricultural Science and Technology, 2020.

[101] R. Ramesh and Kathleen M. Brown. An efficient four-phase heuristic for the generalized orienteering problem. *Computers and Operations Research*, 18(2):151–165, 1991.

[102] Abhijeet Ravankar, Ankit A. Ravankar, Michiko Watanabe, Yohei Hoshino, and Arpit Rawankar. Development of a low-cost semantic monitoring system for vineyards using autonomous robots. *MDPI Agriculture*, 10(5):182:1–182:19, 2020.

[103] David Reiser, El-Sayed Sehsah, Oliver Bumann, Jrg Morhard, and Hans W. Griepentrog. Development of an autonomous electric robot implement for intra-row weeding in vineyards. *MDPI Agriculture*, 9(1):18:1–18:12, 2019.

[104] Hasnaa Rezki and Brahim Aghezzaf. The bi-objective orienteering problem with budget constraint: Grasp-ils. In *IEEE International Colloquium on Logistics and Supply Chain Management*, pages 25–30, 2017.

[105] Giuseppe Riggio, Cesare Fantuzzi, and Cristian Secchi. A low-cost navigation strategy for yield estimation in vineyards. In *IEEE International Conference on Robotics and Automation*, pages 2200–2205, 2018.

[106] Jose Luis Susa Rincon, Pratap Tokekar, Vijay Kumar, and Stefano Carpin. Rapid deployment of mobile robots under temporal, performance, perception, and resource constraints. *IEEE Robotics and Automation Letters*, 2(4):2016–2023, 2017.

[107] Juan Jess Roldn, Jaime del Cerro, David Garzn-Ramos, Pablo Garcia-Aunon, Mario Garzn, Jorge de Len, and Antonio Barrientos. *Service Robots*, chapter Robots in Agriculture: State of the Art and Practical Experiences, pages 67–90. IntechOpen, 2017.

[108] Pedro Santana, Sylvie Thiebaux, and Brian Williams. Rao*: An algorithm for chance-constrained pomdp's. *AAAI Conference on Artifical Intelligence*, pages 3308–3314, 2016.

[109] Lus Santos, Filipe Neves dos Santos, Jorge Mendes, Nuno Ferraz, Jos Lima, Raul Morais, and Pedro Costa. Path planning for automatic recharging system for steep-slope vineyard robots. In *Iberian Robotics Conference*, pages 261–272, 2017.

[110] Lus C. Santos, Filipe N. Santos, E. J. Solteiro Pires, Antnio Valente, Pedro Costa, and Sandro Magalhes. Path planning for ground robots in agriculture: A short review. In *IEEE International Conference on Autonomous Robot Systems and Competitions*, pages 61–66, 2020.

[111] G.D. Schaible and M.P. Aillery. *Challenges for US Irrigated Agriculture in the Face of Emerging Demands and Climate Change*, chapter Competition for Water Resources 2.1.1, pages 44–79. Elsevier, 2017.

[112] Michael Schilde, Karl F. Doerner, Richard F. Hartl, and Guenter Kiechle. Metaheuristics for the bi-objective orienteering problem. *Swarm Intelligence*, 3:179–201, 2009.

[113] Linn I. Sennott. Constrained average cost markov decision chains. *Probability in the Engineering and Informational Sciences*, 7(1):69–83, 1993.

[114] Abhisesh Silwal, Joseph R. Davidson, Manoj Karkee, Changki Mo, Qin Zhang, and Karen Lewis. Design, integration, and field evaluation of a robotic apple harvester. *Journal of Field Robotics*, 34(6):1140–1159, 2017.

[115] Franscesco Betti Sorbelli, Stefano Carpin, Federico Cor, Alfredo Navarra, and Cristina Pinotti. Optimal routing schedules for robots operating in aisle-structures. In *IEEE International Conference on Robotics and Automation*, pages 4927–4933, 2020.

[116] Thomas C. Thayer and Stefano Carpin. Solving large scale stochastic orienteering problems with aggregation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2452–2458, 2020.

[117] Thomas C. Thayer and Stefano Carpin. An adaptive method for the stochastic orienteering problem. *IEEE Robotics and Automation Letters*, 6(2):4185–4192, 2021.

[118] Thomas C. Thayer and Stefano Carpin. A fast algorithm for stochastic orienteering with chance constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2021. (To Appear).

[119] Thomas C. Thayer and Stefano Carpin. A resolution adaptive algorithm for the stochastic orienteering problem with chance constraints. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2021. (To Appear).

[120] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Multi-robot routing algorithms for robots operating in vineyards. In *IEEE International Conference on Automation Science and Engineering*, pages 14–21, 2018. Best Paper Award.

[121] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Routing algorithms for robot assisted precision irrigation. In *IEEE International Conference on Robotics and Automation*, pages 2221–2228, 2018.

[122] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Bi-objective routing for robotic irrigation and sampling in vineyards. In *IEEE International Conference on Automation Science and Engineering*, pages 1481–1488, 2019.

[123] Thomas C. Thayer, Stavros Vougioukas, Ken Goldberg, and Stefano Carpin. Multirobot routing algorithms for robots operating in vineyards. *IEEE Transactions on Automation Science and Engineering*, 17(3):1184–1194, 2020.

[124] Paolo Toth and Daniele Vigo, editors. *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, 2002.

[125] David Tseng, David Wang, Carolyn Chen, Lauren Miller, William Song, Joshua Viers, Stavros Vougioukas, Stefano Carpin, Juan Aparicio Ojea, and Ken Goldberg. Towards automating precision irrigation: Deep learning to infer local soil moisture conditions from synthetic aerial agricultural images. In *IEEE International Conference on Automation Science and Engineering*, pages 284–291, 2018.

[126] Theodore Tsiligirides. Heuristic methods applied to orienteering. *Journal of Operational Research Society*, 35(9):797–809, 1984.

[127] James P. Underwood, Calvin Hung, Brett Whelan, and Salah Sukkarieh. Mapping almond orchard canopy volume, flowers, fruit, and yield using lidar and vision sensors. *Computers and Electronics in Agriculture*, 130:83–96, 2016.

[128] Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. A guided local search metaheuristic for the team orienteering problem. *European Journal of Operational Research*, 196:118–127, 2009.

[129] Pieter Vansteenwegen, Wouter Souffriau, Greet Vanden Berghe, and Dirk Van Oudheusden. *Metaheuristics in the Service Industry*, volume 624, chapter Metaheuristics for Tourist Trip Planning, pages 15–31. Springer Berlin Heidelberg, 2009.

[130] Pieter Vansteenwegen, Wouter Souffriau, and Dirk Van Oudheusden. The orienteering problem: A survey. *European Journal of Operational Research*, 209(1):1–10, 2010.

[131] Pradeep Varakantham and Akshat Kumar. Optimization approaches for solving chance constrained stochastic orienteering problems. In *International Conference on Algorithmic Decision Theory*, pages 387–398, 2013.

[132] Pradeep Varakantham, Akshat Kumar, Hoong Chuin Lau, and William Yeoh. Risk-sensitive stochastic orienteering problems for trip optimization in urban environments. *Transactions on Intelligent Systems and Technology*, 9(3):24:1–24:25, 2018.

[133] Aleixandre Verger, Nathalie Vigneau, Corentin Chron, Jean-Marc Gilliot, Alexis Comar, and Frdric Baret. Green aera index from an unmanned aerial system over wheat and rapeseed crops. *Remote Sensing of Environment*, 152:654–664, 2014.

[134] Thibaut Vidal, Gilbert Laporte, and Piotr Matl. A concise guide to existing and emerging vehicle routing problem variants. *European Journal of Operational Research*, 286:401–416, 2020.

[135] Stavros G. Vougioukas. Agricultural robots. *Annual Review of Control, Robotics, and Autonomous Systems*, 2:365–392, 2019.

[136] Marie Weiss, Frdric Jacob, and Grgory Duveiller. Remote sensing for agricultural applications: A meta-review. *Remote Sensing of Environment*, 236:1–19, 2020.

[137] Xu. Wenzheng, Zichuan Xu, Jian Peng, Weifa Liang, Tang Liu, Xiaohua Jia, and Sajal k. Das. Approximation algorithms for the team orienteering problem. In *IEEE Conference on Computer Communications*, pages 1389–1398, 2020.

[138] Henry A.M. Williams, Mark H. Jones, Mahla Nejati, Matthew J. Seabright, Jamie Bell, Nicky D. Penhall, Josh J. Barnett, Mike D. Duke, Alistair J. Scarfe, Ho Seok Ahn, JongYoon Lim, and Bruce A. MacDonald. Robotic kiwifruit harvesting using machine vision, convolutional neural networks, and robotic arms. *Biosystems Engineering*, 181:140–156, 2019.

[139] Jun Xu and Chengcheng Guo. Optimal mode selection in d2d communication with deadline constraint: A cmdp perspective. *IEEE Wireless Communications Letters*, 9(8):1245–1248, 2020.

[140] Wenzheng Xu, Weifa Liang, Zichuan Xu, Jian Peng, Dezhong Pen, Tang Liu, Xiaohua Jia, and Sajal K. Das. Approximation algorithms for the generalized team orienteering problem and its applications. *IEEE/ACM Transactions on Networking*, 29(1):176–189, 2021.

[141] Jingjin Yu. Intracrtability of optimal multirobot path planning on planar graphs. *IEEE Robotics and Automation Letters*, 1(1):33–40, 2016.

[142] Jingjin Yu and M. LaValle. Optimal multiroot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5):1163–1177, 2016.

[143] Umar Zangina, Salinda Buyamin, M.S.Z. Abidin, and M.S.A. Mahmud. Agricultural rout planning with variable rate pesticide application in a greenhouse environment. *Alexandria Engineering Journal*, 60:3007–3020, 2021.

[144] Yawei Zhang, Yunxiang Ye, Zhaodong Wang, Matthew E. Taylor, Geoffrey A. Hollinger, and Qin Zhang. Intelligent in-orchard bin-managing system for tree fruit production. In *IEEE International Conference on Robotics and Automation*, pages 1–4, 2015.