Improving operational autonomy for unmanned aerial vehicles

Βελτίωση επιχειρησιακής αυτονομίας για μη επανδρωμένα εναέρια οχήματα

Polychronis Georgios

Supervisor: Assoc. Prof. Lalis Spyros

2nd committe member: Assist. Prof. Katsaros Dimitrios

A Thesis submitted in fulfillment of the requirements
for the degree of Diploma Thesis

in the

Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

October 2018

Dedicated to

family and friends

Βελτίωση επιχειρησιακής αυτονομίας για μη επανδρωμένα εναέρια οχήματα

Περίληψη

Στην παρούσα εργασία εξετάζουμε μία παραλλαγή του Vehicle routing problem. Έχοντας ένα σύνολο από σημεία ενδιαφέροντος, θέλουμε να επισκευτούμε αυτά χρησιμοποιώντας ένα όχημα με περιορισμένο ενεργειακό απόθεμα, αλλά με τη δυνατότητα επαναφορτισμού σε συγκεκριμένους κόμβους. Επίσης υποθέτουμε ότι το κόστος κίνησης του οχήματος, καθώς και ο επαναφορτισμός του αποτελούν τυχαίες μεταβλητές. Καθώς η επίσκεψη όλων των κόμβων ενδιαφέροντος μπορεί να μην είναι εφικτή, στοχεύουμε στη μεγιστοποίηση του αριθμού των κόμβων ενδιαφέροντος που θα επισκευτούμε. Προτείνουμε κάποιες ευρετικές λύσεις για το παρόν πρόβλημα και συγκρίνουμε την επίδοσή τους με ενδεικτικούς αλγορίθμους αναφοράς.

Improving operation autonomy for unmanned aerial vehicles

Abstract

This work focuses in a variation of the vehicle routing problem (VRP). The goal is to visit a set of target nodes using a vehicle with energy restrictions but also with the ability to refuel/recharge in some depot nodes. We also consider the travel costs and gain to be stochastic. As it may not always be possible to visit all of the target locations, we aim at maximizing the number of the locations visited. We are proposing two heuristic solutions and compare them with indicative reference algorithms.

Acknowledgements

I would like to thank my supervisor Professor Lalis Spyros for all his help, support and advice. It was a privilege to have him as professor and supervisor.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

2 Related work **3**

2.1 Vechicle routing with fuel/enery constraints and periodic visits 3

2.2 Vehicle routing with stochastic elements 4

2.3 Energy-efficient path planning 6

3 Model **7**

3.1 Terrain and travel paths 7

3.2 Energy reserves 8

3.3 Energy costs 8

3.4 Energy gains 9

3.5 Configurations and scenarios 9

3.6 Path feasibility 9

3.7 Success metrics 11

4 Algorithmic skeleton **12**

4.1 Complexity 15

5 Budget maximization heuristic **16**

5.1 Plug-in functions 16

5.2 Adapted Bellman-Ford algorithm 18

5.2.1 Adaptations 18
5.2.2 Code 22
5.3 Complexity 25

6 Risk minimization heuristic 26
6.1 Complexity 27

7 Reference algorithms 30
7.1 Hop minimization 30
7.2 Ant colony optimization 30
7.3 Oracle 31

8 Evaluation 32
8.1 Setup 32
8.2 Testing different thresholds 34
8.3 Comparison with MinHops and ACO 35
8.4 Comparison with oracle 37

9 Conclusion 41

List of Figures

5.1 The same cycle ($N2$, $N3$, $N4$, $N2$) can be beneficial (a) or not beneficial (b), depending on the current budget. The edge costs are shown on the respective edges, the node gains are shown inside each node. In both cases, the maximum budget is 10. 20
5.2 For start node $N1$, destination node $N4$, and initial budget is 9, the path that minimize the cost is not the same as the path that maximizes the budget. The edge costs are shown on the respective edges, the node gains are shown inside each node. The maximum budget is 10. 21

8.1 Graph used in the experiments. The start node $(0, 0)$ is in blue and the depot nodes $(2, 5), (8, 6)$ in green. 33
8.2 Effect of $Threshold_{normal}$ on the MaxBudget heuristic. 36
8.3 Effect of $Threshold_{normal}$ on the MinRisk heuristic. 36
8.4 Effect of $Threshold_{optimistic}$ on the MaxBudget heuristic. 37
8.5 Effect of $Threshold_{optimistic}$ on the MinRisk heuristic. 37
8.6 Comparison of our heuristics with ACO and minimizing hops algorithm. 38
8.7 Comparison of our heuristics and the oracle. 40

Chapter 1

Introduction

Unamnned vehicles (UVs) will play a major role in next-generation applications. In particular, unmanned aerial vehicles (UAVs) are already being used in the domain of argiculture and surveillance. Other types of UVs, such as unmanned ground vehicles (UGVs) or umanned underwater vehicles (UUVs), though more exotic, are also becoming more mature, and will most likely be used in several applications in the future.

A typical scenario is for UVs to visit certain points of interest in order to take some measurements or to detect certain objects or phenomena. For example, in agriculture, UVs scan a crop field to detect problematic spots that are infected with pests. Similarly, in search and rescue missions, UVs can scan a wide area to find missing people. Yet another example is for a UV to patrol an area by going through pre-defined locations. In all these cases, it is desired for the UV to perform the mission as efficiently as possible.

This problem is known as the *vehicle routing problem* (VRP), which in turn is an extension of the *travelling salesman problem* (TSP). In a nutshell, the VRP consists in finding a plan that can be followed by a vehicle to visit all the target locations. The problem has been widely studied with many variations on different constrains, e.g., regarding the specific paths that can be followed by the vehicle, the locations to be visited, or the time when certain locations have to be visited. Also, in several formulations, the vehicle is assumed to have finite fuel/energy reserves or serving capacity, which can be replenished by visiting special depot nodes.

We investigate a special variant of the VRP problem where there is some uncertainty regarding the cost of travel and the energy harvesting / refueling opportunities of the depot nodes. Algorithms that compute the travel plan based on static data are not suitable in this case, because these plans may actually turn out to be infeasible. Instead, a more dynamic approach is required, where the system state is reviewed en route and the routing decisions are adjusted accordingly, at runtime.

The rest of the thesis is structured as follows. Chapter 2 gives an overview of related work. Chapter 3 describes the system model and the problem we study in a more formal way. Chapter 4 gives a common high-level skeleton/pattern that is followed by all the proposed heuristics. Then, two different heuristic algorithms are presented in Chapter 5 and Chapter 6, respectively. As a reference, we use two more algorithms, described in Chapter 7. In Chapter 8, we evaluate the algorithms for a range of different scenarios. Finally, Chapter 9 concludes the thesis.

Chapter 2

Related work

Different variants of the VRP and the TSP have already been investigated. Next, we give a brief overview, and compare with the problem we address in our work.

2.1 Vechicle routing with fuel/enery constraints and periodic visits

In the so-called *periodic vehicle routing problem* and *multi-depot periodic vehicle routing problem* [1,2,6,8,23], the objective is to periodically visit a set of destinations (target nodes), using one or multiple vehicles. Vehicles have limited energy reserves, and may recharge/refuel at special depot stations. There are more variations where the targets have different priorities, vehicles have different capabilities/capacities, and the targets must be visited within a time interval. We briefly dicuss such work below.

Work in [17] investigates the problems of *persistent visitation with fuel constraints*. In this case, the vehicle perpetually visits the target nodes (customer nodes), while each customer must be visted at a specific rate. The fuel of the vehicle is limited, so it is necessary to visit nodes with refueling properties. The problem aims at satisfying the rate of visit for each destination, as well as at minimizing the total cost of fuel consumption.

The probem of *continued monitoring* is studied in [18], where the mission is accomplished by several heterogeneous vehicles of different types with limited re-

sources. The objective is to visit periodically a set of target nodes with different priorities in a given time interval. The rate at which each target node requires a visit (by any of the vehicles) is proportional to its priority. There is also a set of depot nodes where vehicles can change their batteries, however each type of vehicle requires a different type of battery; only vehciles of the same type can use the same type of battery. Also, the number of the different types of batteries available at each depot is limited.

All the above formulations correspond to VRPs with a fleet of one or more vehicles that have energy/fuel limitations. Each edge of the graph is associated with a cost, and the energy of the vehicles decreases accordingly based on that cost. There are also one or more so-called depot nodes, where the vehicles can refuel, fully or partially. The main goal is to find a path that minimizes the total cost of the mission, which is the sum of the costs of the edges chosen to be used in the path of the vehicles. Another objective is to satisfy the visit rates of the target nodes.

The main difference between the above and our work is that in our case the cost of travel between nodes as well as the amount of energy/fuel that can be gained at depot nodes, is not known with full certainty.

2.2 Vehicle routing with stochastic elements

The objective of *vehicle routing problem with stochastic demands* is for a fleet of one or more vehicles, which have a finite serving capacity and may also have fuel constraints, to visit and serve a set of known target nodes (customers). However, the exact demand of each customer is not known beforehand. As a result, when a vehicle visits a customer, it may find out that the demand exceeds its remaining capacity. The failure to properly service the customer can be ignored, at a penalty, and the vehicle may proceed to the next customer. Aternatively, the vehicle may go to a depot node in order to acquire the missing resources to restore its servicing capacity, and then return to service the customer. This problem is researched in many works [5, 16, 20]. Moreover, in [7] the problem is investigated for the more general case where the goal is to minimize the travel time with the additional constraint of

keeping the travel time lower than a given upper bound.

In the *vehicle routing problem with stochastic travel times*, a fleet of one or more vehicles, which may have capacity and/or fuel constraints, visit a set of known targets, but in this case, the travel time is a random variable. Also, each target has a time window when it is available for visits by a vehicle. If the target is not visited/served within that time window, there is a penalty. The objective is to route the vehicles so as to minimize the total penatly. Models like this are described in [14, 15, 21, 22].

The *vehicle routing problem with stochastic customers* consists in using a fleet of one or more vehicles, which may have capacity and/or fuel constraints, to visit a set of potential target customers. However, these targets require a visit only with some probability. As a result, a vehile may visit a node that does not need to be serviced. Indicative work can be found in [4, 9, 10].

Our work is more similar to the VRP with stochastic travel times. However, in this case the main constraint is the time, whereas in our work the main constraint is the energy of the vehicle, which can also increase its energy reserves by visiting certain nodes. It is also important to note that in our work the mission ends when the vehicle exhausts its energy. This is a major difference compared to the VRP with stochastic demands, in which case the vehicle can always go to a depot node in order to restore its capacity. As a consequence, in our work, it is not always possible to visit all nodes, in contrast to most other problem formulations where this is always feasible and the problem consists in minimizing the travel cost and/or time violation penalty. In this sense, our work is closer to [7], which has a similar constraint for the travel cost. But in our case the stochasticity concerns the travel cost and enegy gains, not the customer demands. The difference between this work and ours is that we have stochastic travel costs and energy gains, as well as that our goal is to maximize the number of target nodes that are visited.

2.3 Energy-efficient path planning

Several algorihtms have been designed to compute an energy-efficient route between two nodes, based on an abstract graph where the weights at edges represent the energy consumption or the energy restoration. The difference to a typical shortest-path algorithm, is that the computed routes minimize the sum of edge costs, not the number of edges. For instance, [19] deals with the problem of energy-efficiency as a special case of the constrained shortest path problem, while in [3] the authors see the problem of energy-efficient path planning as a cost minimization problem. In both cases, the ability of the vehicle to gain energy / recharge is modeled by introducing edges with negative weights.

Our work is also related to the problem of findind a cost-effective path between two nodes. However, the mimimization of the the cost of travel per se is not as central as in the above algorithms, because it can be counter-balanced by gaining energy at depot nodes. What is ultimately important is for the vehicle to take a route that turns out to be feasible, while visiting as many target nodes as possible. To this end, one of our heuristics use (as a component) a suitably adapted version of the Bellman-Ford algorithm [11], which finds the path between two nodes that maximizes the energy reserves of the vehicle (as opposed to minimizing the travel cost). Notably, our version allows beneficial cycles, which are not allowed in the original algorithm since this would mean that the travel cost can be reduced infinitely.

Chapter 3

Model

This section briefly presents the system model, and defines the problem we study in a more formal way.

3.1 Terrain and travel paths

The terrain of travel is modeled as an incomplete directed graph (N, E), where N is the set of nodes and E is the set of edges between nodes. Each node $n_i \in N$ represents a distinct geographical location in the terrain. Each edge $e_{i,j} \in E$ denotes the fact that the vehicle can move directly from node n_i to node n_j without having to go via one or more intermediate nodes.

Edges have a direction and there can be at most one edge between two nodes per direction. The fact that it is possible to move directly from n_i to n_j does not mean that this is possible in the reverse direction. In other words, $e_{i,j} \in E \;\not\Longrightarrow\; e_{j,i} \in E$.

The vehicle can vist nodes by travelling across the edges of the graph. The travel path of the vehicle is encoded as the sequence of nodes that are visited by the vehicle, including the start and end node of the path. Let $p_{k_1,k_2,k_3,..,k_{m-1},k_m}$ denote the path that starts from node n_{k_1} and goes through n_{k_2}, n_{k_3} etc to end at node n_{k_m}, corresponding to the list of edges $(e_{k_1,k_2}, e_{k_2,k_3}, ..., e_{k_{m-1},k_m})$. Let $occ(p, e_{i,j})$ denote the number of times edge $e_{i,j}$ occurs in p, and $occ(p, n_i)$ be the number of times node n_i occurs as the starting point of an edge in p. Also, let $p \oplus e_{i,j}$ denote the path that results by appending edge $e_{i,j}$ to p. Finally, let $nodes(p)$ denote the set

of nodes in p (a node n_i will appear only once in this set even if it appears multiple times in the path), and let $len(p)$ denote the number of edges or hops in p.

The set of nodes $V \subseteq N$ includes all the nodes that correspond to the target locations that have to be visited by the vehicle. We are interested in algorithms that guide the movement of the vehicle so that the vehicle visits all nodes in V. Note that the vehicle may be required to start its journey from node $n_s \notin V$, and may be required to end its journey at node $n_d \notin V$. Also, the path it will follow may include more nodes than the ones in V. So, in the general case, the end result can be a path $p_{s,k_1,k_2,k_3,..,k_m,d}$ so that $V \subseteq nodes(p)$.

3.2 Energy reserves

We assume that the vehicle has limited energy storage capacity B_{max}. This can be thought of as the capacity of a fuel tank or the capacity of a battery. Let the current energy budget (reserves) of the vehicle be denoted by b. Obviously, $b \leq B_{max}$ holds at any point in time.

3.3 Energy costs

The movement of the vehicle comes at a cost. Let $c_{i,j}$ denote the cost incurred when the vehicle travels from n_i to n_j over $e_{i,j}$, also referred to as edge cost.

The edge cost is *not known* in advance with certainty. However, we assume that the edge cost $c_{i,j}$ follows a *known* random distribution over the range $[c_{i,j}^{min}..c_{i,j}^{max}]$ with an expected/mean value of $c_{i,j,}^{mean}$.

Without loss of generality, we assume that the edge cost is equal to the amount of energy that needs to be spent by the vehicle in order to perform this movement. If the vehicle has an energy budget b and moves from n_i to n_j over $e_{i,j}$, the remainining budget will be $b' = b - c_{i,j}$. If $b' \leq 0$, the vehicle will exhaust its energy and end its journey before reaching n_j.

3.4 Energy gains

The vehicle may increase its energy budget, by gaining some energy at certain nodes. One can think of theses nodes as refueling/recharging stations or geographical areas where the vehicle may gain some dynamic energy.

Similarly to the edge costs, the amount of energy that can be gained at node n_i is a random variable g_i, which follows a *known* random distribution range $[g_i^{min}..g_i^{max}]$ with an expected/mean value of g_i^{mean}.

It may be known in advance that certain nodes cannot be used by the vehicle to gain some energy. In this case, $g_i^{min} = g_i^{max} = 0$, hence these nodes have zero probability for energy gains. Let the set of nodes $G = \{n_i|g_i^{min}, g_i^{max} > 0\}$ include all the nodes with a non-zero probability of energy gain. In the general case, $V \cap G \neq \emptyset$, in other words it is possible for a node which the vehicle has to visit to also offer some energy gain opportunities.

3.5 Configurations and scenarios

We refer to a given graph (N, E) with a given random distribution for each edge cost $c_{i,j}$ and a given random distribution for each node gain g_i, as a *configuration*. As noted, the actual edge costs $c_{i,j}$ and node gains g_i are not a priori known, and may change each time the vehicle crosses edge $e_{i,j}$ or visits node n_i, respectively.

To capture the ground truth, we refer to the actual values of edges costs and node gains, as *scenarios*. Of course, the scenario is not known to the vehicle (but it can be known to an oracle path planner). For a given scenario, we let $c_{i,j}^k$ denote the actual cost of edge $e_{i,j}$ when the vehicle crosses that edge for the kth time. Similarly, g_i^k is the gain of node n_i when the vehicle visits that node for the kth time.

3.6 Path feasibility

In order for a planned path to be *feasible*, the budget of the vehicle must be sufficient to cover the cost for crossing each edge along that path. In the following, we capture this constraint in a more formal way.

First, we capture the budget that remains available when starting with an initial budget b and performing a hop from n_i to n_j over edge $e_{i,j}$:

$$rem_{1hop}(b, cp, p_{i,j}) = min(B_{max}, b + g^{occ(cp,n_i)+1}) - c_{i,j}^{occ(cp,e_{i,j})+1} \tag{3.6.1}$$

where cp (for current path) is the path that has already been followed by the vehicle until this point. In words, the node gain of n_i is added to the budget b (up to the maximum energy storage capacity B_{max}) and then the edge cost is substracted in order for the vehicle to cross the edge that leads from n_i to n_j. Note the usage of $occ()$ in order to take into account previous occurances of node n_i and edge $e_{i,j}$ in the path that has already been followed by the vehicle. The equation does not take into account the energy gain at the destination node n_j (if any), as this cannot be exploited in order to cross the edge $e_{i,j}$.

Based on the above equation, we can define the remaining budget for the general case of a multi-hop path p from node n_i to node n_j, using a recursive formula:

$$rem(b, cp, p_{i,k_1,k_2,\ldots,k_m,j}) = \begin{cases} rem_{1hop}(b, cp, p_{i,j}) & m = 0 \\ rem(rem(b, cp, p_{i,k_1}), cp \oplus e_{i,k_1}, p_{k_2,\ldots,k_m,j}) & m > 0 \end{cases} \tag{3.6.2}$$

In words, the budget that remains after taking a multi-hop path equals the remainining budget for the path without the first hop, starting with a budget that is equal to the remaining budget after taking the first hop. As this is the case for the 1-hop path, the remaining budget of a multi-hop path does not include the gain of the destination node.

We can then define the feasibility of a planned path, if the vehicles has already travelled along the path cp and has a current budget of b, as follows. We say that $p_{k_1,k_2,\ldots,k_m}$ is feasible if all prefix paths $p_{k_1,k_2,\ldots,k_x}$, $1 < x \leq m$ (including the full path itself) have a positive remaining budget. More formally:

$$feasible(b, cp, p_{k_1,k_2,\ldots,k_m}) \equiv rem(b, cp, p_{k_1,k_2,\ldots k_x}) > 0 : \forall x : 1 < x \leq m \tag{3.6.3}$$

It is obvious that $feasible(b, cp, p) \implies rem(b, cp, p) > 0$. In other words, if a path is feasible then the budget will not be completely exhausted at the destination node. But, if p is a multi-hop path, the reverse does not hold, $rem(b, cp, p) > 0 \not\Longrightarrow$

$feasible(b, p)$. This is because the fact that the remaining budget is positive at the end of a path does not guarantee that it will not be (temporarily) exhausted at some point along that path.

3.7 Success metrics

We wish to find an algorithm that guides the vehicle so that it manages to visit all nodes in V without exhausting its budget while en route. In other words, starting with an initial budget b, it is desired to find a path $p_{k_1,k_2,,..,k_m}$ so that $V \subseteq nodes(p)$ and $feasible(b, p)$. We call such paths *perfect* solutions.

Note that for certain system configurations and/or scenarios, a perfect solution may not even exist. Neverthelss, imperfect solutions are not a total failure, and some imperfect solutions can be clearly better than others. To have a more general metric, we introduce the so-called *success ratio* of a path p as the ratio between the number of the nodes of interest that are part of the path that was followed by the vehcile and the total number of nodes of interest that the vehicle should visit: $success(p, V) = |nodes(p) \cap V|/|V|$.

Chapter 4

Algorithmic skeleton

In this section, we describe a common algorithmic skeleton that is followed by all the proposed heuristics. The input to the algorithm is the graph (N, E), the set V of the nodes to visit, the node n_s from where the patrol starts, and the initial energy budget b of the vehicle (without the gains of the start node).

Our heuristics work according to a specific pattern, as follows. In a first step, a path is chosen from the source node to some node of interest. In a second step, the vehicle tries to follow that path, hop by hop. After each hop, an assessment of the current situation is made, and this is compared to the assumptions of the path planning step. If things go more or less according to plan, the vehicle continues its journey along the current path, else it stops following the current path and a fresh path planning step is performed, based on the situation at hand. When a node of interest is reached, this is removed from the set of nodes to visit, and the algorithm continues for the rest of the nodes, by planning and then following a path to another node of interest.

In the planning phase, the paths are chosen based on *estimated* values for the edge costs and node gains. Two different estimation modes are employed. In the *normal* mode, these estimates correspond to the expected/mean values ($c_{i,j}^{mean}$ and g_i^{mean}) of the respective random distributions. In the *optimistic* mode, the estimates used for the edge costs are equal to the minimum values ($c_{i,j}^{min}$) of the respective distributions, while the estimates for the node gains are the maximum (g_i^{max}) values of the respective distributions. In each iteration, first, an attempt is made to find a

path using the normal mode. If this does not lead to a feasible path, the search is repeated, for a second time, in the optimistic mode. If this fails then the algorithm has arrived at a dead end — it is certain that there is no feasible path from the current location of the vehicle to any of the remaining nodes of interest. In this case, the algorithm terminates without the vehicle having visited all nodes of interest.

This algorithmic pattern is captured in the form of a generic skeleton, given in Algorithm 1. The functionality of the different heuristics, which will be discussed in the following sections, is captured via two plug-in functions that have different implementations depending on the rationale of the heuristic.

More concretely, the plug-in function $ChoosePath(n_s, b, V, mode)$ contains the logic that encodes the core heuristic. It takes as parameters the node n_s where the vehicle is currently located, the current available/remaining budget b, the set of nodes V that (still) have to be visited and the edge cost / node gain estimation mode *mode*, and returns the path to folow. It is important to note that the path that is returned by $ChoosePath()$, is a *complex* data structure p, where the individual hops are recorded as an array $p.hops[k], 0 \leq k \leq len(p)$, and where $p.hops[k].n$ is the node at the kth hop of the path; more specifically, $p.hops[0].n$ is the start node of p (this is always equal to node n_{cur} that was supplied as a parameter), and $p.hops[len(p)].n$ is the end node of the path.

The estimates used in $ChoosePath()$ for the edge costs and node gains to pick the preferred path (in the first phase of the algorithm), may prove wrong in reality, when the vehicle actually tries to follow the suggested path (in the second phase of the algorithm). For this reason, function $UnexpectedOutcome()$ is used to check if the outcome turns out to be (significantly) different than what was expected in $ChoosePath()$. If things do not go as planned, the plan is updated by invoking once again the $ChoosePath()$ function. A new path is also chosen when the current path is successfully followed to its end, and there are still some nodes of interest that have not yet been visited.

Note that, depending on the heuristic, the path data structure can be extended to include additional information. In the same vein, the skeleton can be extended to keep track of additional information, which can be passed as an additional parameter

Algorithm 1 Skeleton for the heuristics

```
function SKELETON(node set V, node n_s, int budget)
    node n_cur ← n_s                                        ▷ current node
    int b ← budget                                          ▷ remaining budget
    node set remV ← V ⊖ n_cur                               ▷ nodes to visit
    path p ← null                                           ▷ path to follow
    while remV ≠ ∅ do
        b ← min(b + g_cur, B)                               ▷ exploit node gain
        if p = null then
            p ← ChoosePath(n_cur, b, remV, NORMAL)
            if p = null then
                p ← ChoosePath(n_cur, b, remV, OPTIMISIC)
                if p = null then                            ▷ dead end, terminate
                    return remV
                end if
            end if
            k ← 1                                           ▷ set path hop counter
        end if
        n_nxt ← p.hops[k].n                                 ▷ hop to next node of the path
        b ← b − c_cur,nxt                                   ▷ pay travel cost for edge e_cur,nxt
        if b < 0 then                                       ▷ budget exhausted, terminate
            return remV
        end if
        n_cur ← n_nxt
        remV ← remV ⊖ n_cur
        if k = len(p) ∨ UnexpectedOutcome(p, k, b, ..., mode) then
            p ← null                                        ▷ plan new path
        else
            k ← k + 1                                       ▷ proceed with the next hop
        end if
    end while
    return remV
end function
```

to the $UnexpectedOutcome()$ function.

4.1 Complexity

The complexity of the skeleton depends on the complexity of the path finding heuristic, implemented in $ChoosePath()$, and the number of times this is invoked in order for the vehicle manages to visit all nodes of interest.

In the best case, the path heuristic will be invoked $|V|$ number of times, once for each node of interest, yielding a complexity of $O(|V| \times P)$ where P is the complexity of the path heuristic. For the case where $|V| << N$, the complexity is dominated by the path heursitic and becomes equal to $O(P)$.

In the worst case, the path heuristic will be performed after every single hop performed, in both modes. Assuming that the length of each path followed to reach a node of interest is in the order of magnitude of the graph diameter D, the complexity is $O(|V| \times D \times 2 \times P)$. Again, if $|V| << N$, the comlexity becomes $O(D \times P)$.

Chapter 5

Budget maximization heuristic

One heuristic is to favor paths that lead to nodes of interest while maximizing the remaining budget. The intuition is that the budget that remains available after visiting a node of interest can be further exploited to pursue more paths and to visit more nodes of interest in the future.

5.1 Plug-in functions

The code for the respective plug-in functions is given in Algorithm 2.

Function $ChoosePath(n_s, budget, V, mode)$ is implemented as follows. In a first step, a path is found from the source node n_s and an initial budget *budget* to each other node $n_i \in N$ so that this maximizes the remaining budget. This logic is abstracted in a separate function $FindMaxBudgetPathsToAll()$, which is discussed separetely in the sequel. Then, in a second step, from all the paths that lead to some node of interest $n_v \in V$, the one that maximizes the remaining budget at the end of the path is chosen. Ties are broken by giving preference to shorter paths.

The $ChoosePath()$ function returns an array of paths, one path $p[n_v]$ for each $n_v \in N$. Each path $p[n_v]$ has the structure discussed Section 4. In addition, the path data structure is extended to store in $p[n_v].hops[k].b$ the *estimated* available/remaining budget after the kth hop (without taking into account any potential gain at $p[n_v].hops[k].n$). For notational convenience, we let the available/remaining budget at the end of the path be also stored at the level of the entire path $p[n_v].b =$

Algorithm 2 Plug-in functions for the MaxBudget heuristic

function ChoosePath(**node** n_s, **int** $budget$, **node set** V, **string** mode)

    node $n_d \leftarrow null$ ▷ preferred destination node

    int $maxb \leftarrow -\infty$ ▷ remaining budget for n_d

    array of path $p[] \leftarrow FindMaxBudgetPathsToAll(n_s, budget, mode)$

    for $n_v \in V$ **do**

        if $p[n_v] \neq null$ **then**

            if $p[n_v].b > maxb \vee p[n_v].b = maxb \wedge len(p[n_v]) < len(p[n_d])$ **then**

                $maxb, n_d \leftarrow p[n_v].b, n_v$

            end if

        end if

    end for

    if $n_d = null$ **then**

        return null

    else

        return $p[n_d]$

    end if

end function

function UnexpectedOutcome(**path** p, **int** k, **int** $budget$, **string** $mode$)

    if $mode =$ NORMAL **then**

        return $\frac{|p.hops[k].b - budget|}{p.hops[k].b} > Threshold_{normal}$

    else

        return $\frac{|p.hops[k].b - budget|}{p.hops[k].b} < Threshold_{optimistic}$

    end if

end function

$p[n_v].hops[len(p[n_v])].b$.

In this heuristic, function $UnexpectedOutcome()$ takes as input the chosen path p that was returned by $ChoosePath()$, the index k corresponding to the hop that was taken, the remaining budget b after that hop (without including the gain at the destination node of the hop, and the mode of the search. It then compares the remaining budget with the estimated remaining budget for that hop as this was calculated by $FindMaxBudgetPathsToAll()$. In the normal mode, the function returns *true* if this difference is larger than a threshold, to trigger a plan update. In contrast, in the optimistic mode, it returns *true* if the difference is smaller than a threshold. The rationale is that if reality proves to be close to an optimistic estimate then it might be possible to plan the rest of travel based on a more realistic/normal estimate. For each estimation mode we use a different threshold, but in both cases the threshold is a pecentage reflecting the relative difference between the expected and the actual remaining budget at the given hop. The selection of these thresholds is discussed in the evaluation section.

5.2 Adapted Bellman-Ford algorithm

Function $FindMaxBudgetPathsToAll()$ is implemented using a suitably adapted version of the Bellmann-Ford (BF) algorithm [11]. BF is designed to find the shortest paths from a given source node to all the other nodes in a graph. More specifically, the edges of the graph are associated with a distance attribute, and the paths found are those that minimize the total distance between the source and the destination node. In our case, we use BF to find the *most beneficial* paths (the ones that maximize the budget) between a given source node and all other nodes in the graph. To this end, we adapt the algorthm to our needs, as explained below.

5.2.1 Adaptations

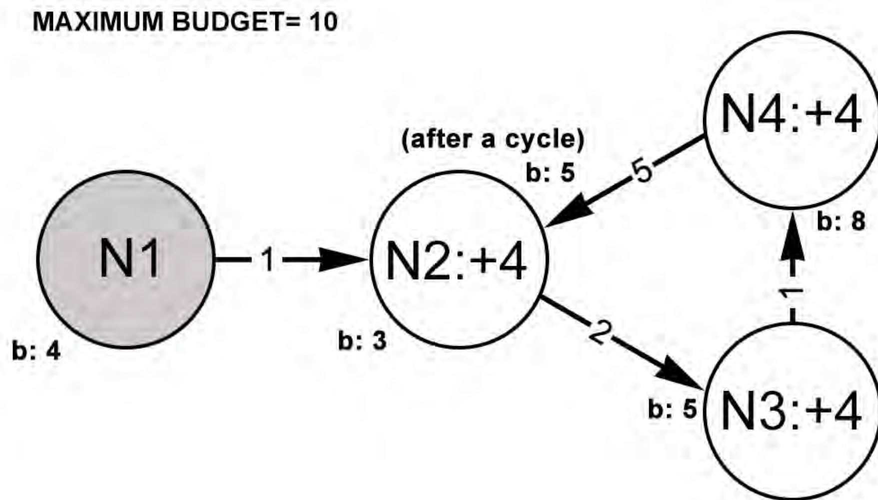
The notion of distance in BF can be mapped directly to the notion of edge cost in our case. However, in BF, the edge distance values are fixed, whereas in our case the edge costs (as well as the node gains) are random variables. Therefore, we use

estimated fixed values. As discussed in Section 4, the values chosen depend on the mode used (normal or optimistic).

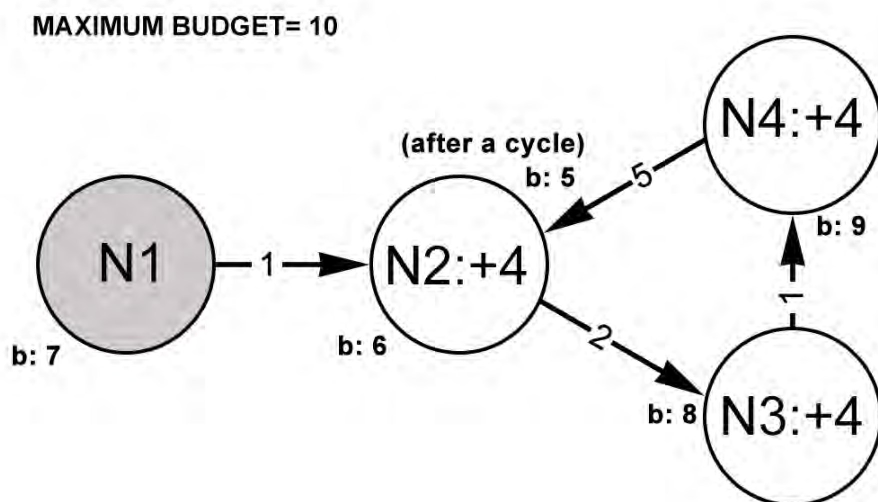
In BF, so-called negative cycles whose edges sum to a negative value, are considered invalid. If such a path is found, BF, exits with error. While in our case negative edges do not exist (recall that $c_{i,j} \geq 0$), energy gains at nodes can have the same effect because they can increase the budget, leading to so-called *beneficial* cycles where the budget at the end of the cycle is greater than the budget in the begining of the cycle. Unlike BF, in our case, beneficial cycles are allowed. In fact, letting the vehicle peform such cycles may be *necessary* in order to complete the mission/patrol successfully. Therefore, instead of exiting, we change the algorithm to accept such cycles and continue as usual. However, note that the same cycle can be beneficial or not, depending on the current budget. For example, Figure 5.1a illustrates the case where the cycle $N2$, $N3$, $N4$, $N2$ is beneficial if the vehicle has a budget of 3, which is increased to 5. In contrast, as shown in Figure 5.1b, the same cycle is not beneficial if the vehicle arrives at $N2$ with a budget of 6. Going through the same cycle will decrease the budget to 5.

BF iteratively checks and updates the mimimum distance for reaching a node. We check and update the best budget that can be achieved by reaching a node (provided there is sufficient budget to reach that node). Note that mimimizing the edge costs is not the same as maximizing the budget. Also note that, due to the maximum budget constraint, the remaining budget along a path cannot be caclulated simply by subtracting the sum of edge costs from the sum of the respective node gains. For example, in the graph shown in Figure 5.2, the path from node $N1$ to node $N4$ with a starting budget of 9 that minimizes the cost is via node $N3$ which provides a gain that cannot be fully exploited due to the budget constraint, whereas the path that maximizes the budget is via node $N2$ which provides the same gain that can be exploited to a full extent.

BF associates each node with a predecessor (that corresponds to the previous hop of the path with the smallest distance so far for that node. This is because a path can be determined by following back the predecessors from a destination node back to the source node. In our case, we record the entire path. This is because, in



(a) Cycle is beneficial.



(b) Cycle is not beneficial.

Figure 5.1: The same cycle ($N2$, $N3$, $N4$, $N2$) can be beneficial (a) or not beneficial (b), depending on the current budget. The edge costs are shown on the respective edges, the node gains are shown inside each node. In both cases, the maximum budget is 10.

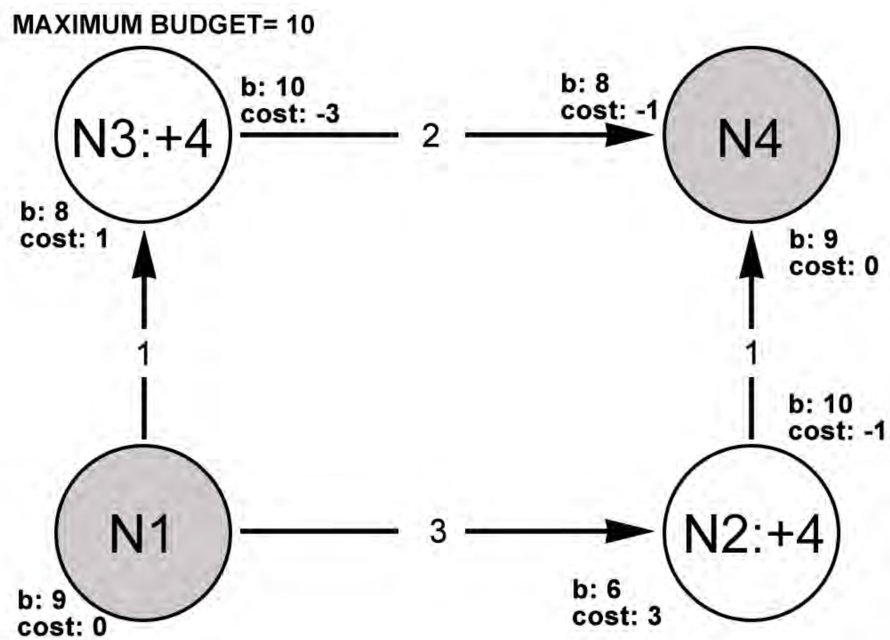


Figure 5.2: For start node $N1$, destination node $N4$, and initial budget is 9, the path that minimize the cost is not the same as the path that maximizes the budget. The edge costs are shown on the respective edges, the node gains are shown inside each node. The maximum budget is 10.

the presence of cycles, knowing the predecessor node is not sufficient to re-construct the path that needs to be followed, as this way one cannot encode the point of entry/exit of a cycle.

BF re-computes the distances and node predecessors $N-1$ times, where N is the number of nodes in the graph. It is guaranteed that this suffices to find the shortest paths from the destination node to all nodes. This is not sufficient in our case, where beneficial cycles may exist and thus paths can be longer than $N-1$ hops. For this reason, we run the algorithm as long as some updates still take place for some nodes.

5.2.2 Code

To show the differences in a more tangible way, Algorithm 3 gives the pseudocode of the original Bellman-Ford algorithm, while the code of the modified version with the adaptations that were discussed above is given in Algorithm 4.

Our adapted code takes as input the starting node n_s and the initial available *budget* of the vehicle (which *does not* include the gain g_s of the starting node). When the search ends, the code returns the most "beneficial path (the path that maximizes the budget) from node n_s to every other $n_i \in N$ and the budget that remains available when taking this path. Again, the remaining budget *does not* include the gain of the end node of the path.

The principle of operation of the adapted version is the same as in the original algoritm. Instead of using the distance, the remaining budget is used as the metric for comparing among different options. Like in the original algorithm, in each iteration the decisive factor for choosing the next edge is the cost. The gain of the source node that can increase the current budget (subject to the budget constraint), applies to all edge options and merely affects the feasibility of certain options.

Note that the algorithm will terminate. Even though beneficial cycles may exist, the same cycle cannot remain beneficial for ever, due to the budget constraint. In other words, there is a finite number of beneficial cycles. Also note that if there are no beneficial cycles, the algorithm works like the original version and finds the optimal path.

Algorithm 3 Bellman-Ford

```
function BellmanFord(n_s)
    node prev[N]                                   ▷ predecessor of each node
    int dist[N]                                    ▷ path distance via predecessor
    for each n_i ∈ N do
        prev[n_i], dist[n_i] ← null, ∞
    end for
    dist[n_s] ← 0
    for 1 ≤ k ≤ |N − 1| do
        for each n_j|e_{i,j} ∈ E do
            tmpdist ← dist[n_i] + c_{i,j}
            if tmpdist < dist[n_j] then
                dist[n_j], prev[n_j] ← tmpdist, n_i
            end if
        end for
    end for
    for each e_{i,j} ∈ E do
        if dist[n_i] + c_{i,j} < dist[n_j] then
            return error                           ▷ negative cycle found
        end if
    end for
    return dist[], prev[]
end function
```

Algorithm 4 Adapted Bellman-Ford algorithm

```
function ADAPTEDBELLMANFORD(node n_s, int budget, string mode)
    path p[N]                                   ▷ array of paths from n_s to every other node
    boolean update                              ▷ flag used for termination
    if mode = NORMAL then                       ▷ use normal estimates
        cmode, gmode ← mean, mean
    else                                        ▷ use optimiztic estimates
        cmode, gmode ← min, max
    end if
    for each n_i ∈ N do
        p[n_i] ← (null, −∞)
    end for
    p[n_s], p[n_s].b ← (n_s, budget), budget
    update ← true
    while update do
        update ← false
        for each n_j|e_ij ∈ E do
            tmpb ← min(p[n_i].b + g_i^gmode, B_max) − c_ij^cmode      ▷ try edge
            if (tmpb > 0) ∧ (tmpb > p[n_j].b) then
                p[n_j], p[n_j].b ← p[n_i]⊕ (n_j, tmpb), tmpb
                update ← true
            end if
        end for
    end while
    return p[]
end function
```

5.3 Complexity

The time complexity of the original Bellman-Ford algorithm is $O(|N| \times |E|)$, where $|N|$ and $|E|$ are the number of nodes and the number of edges, respectively.

In the adapted version of the algorithm, the time complexity also depends on the number of beneficial cycles and the number of the nodes involved in each such cycle. More concretely, assuming K beneficial cycles (including iterations of the same cycle) and an average of M nodes in each cycle, the time complexity of the algorithm is $O((|N| + K \times M) \times |E|)$. In case of no beneficial cycles exist, the time complexity is the same as the original algorithm, $O(|N| * |E|)$.

Chapter 6

Risk minimization heuristic

Another heuristc is to favor paths that minimize the risk of turning out to be infeasible in practice. One metric that captures this risk is the minimum difference between the estimated available budget and estimated edge cost, over all the hops of a path. For brevity, we refer to this metric as the *difference score* or simply path *score.* The lower the difference score of a path, the more likely it is for that path to turn out to be infeasible when the vehicle tries to follow it. So, to minimize the the risk, the difference score has to be maximized.

To this end, the plug-in function $ChoosePath(n_s, budget, V, mode)$ finds a path from the source node n_s to a node $n_v \in V$ so that this path maximizes the difference score. This is implemented as a *best-first* algorithm with a priority queue that contains the path options to be explored. The data structure used to store information for each path p follows the pattern discussed in Section 4. In addition the path data structure is extended so that the *estimated* difference score at the kth hop of the path is stored as $p.hops[k].diff$, and the *estimated* available remaining budget at the last hop of the path is stored in $p.b$. The difference score is used as the priority of the path entries.

The priority queue is initialized with a zero-length path with n_s as the start node, *budget* as the difference score, and *budget* as the available remaining budget. Then, iteratively, as long as the queue is not empty, the next entry is removed (the one with the largest priority) and new entries are added to the queue for each possible hop with appropriately updated fields. When a path to a node of interest is found,

no new entries are added to the queue beyond that point, and all entries with lower scores are removed from the queue —this is because the score of a path can only get lower as more hops are added to it. Also, the path is compared to the best path found so far to a node of interest, and it is used as the best option if it has a higher difference score (again, ties are broken by giving preference to shorter paths). Else, if the path does not lead to a node of interest, all feasible continuations are added in the quue for further explotation. The search stops when the queue becomes empty. If no feasible path is found to any node of interest based on the estimated edge costs and node gains, the function does not return a path.

The code for the respective plug-in functions is given in Algorithm 5.

As in the previous heuristic, in the normal mode, *UnexpectedOutcome*() returns *true* if the metric used (in this case, the difference score) turns out to be significantly different in practice than the respective estimate that was calculated by the $ChoosePath()$ function, whereas in the optimistic mode it returns *true* if the metric turns out to be close to the estimated value. Similarly, the thresholds used apply to the relative difference between the expected and the actual value of the difference score at the given hop.

Note that, in this case too, the skeleton code must be extended accordingly, to track the actual difference score for each hop that is performed by the vehicle. This value is then passed as a parameter to the *UnexpectedOutcome*() function, to be compared with the corresponding estimated value.

6.1 Complexity

An unbounded best-first search has time complexity of $O(b^d)$, where b is the branching factor and d is the distance of the solution from the starting node. Nevertheless, based on empirical data this complexity has a significant difference from the real performance of this algorithm. The real performance is mostly affected by the goodness of the heuristic function, but also by the optimizations that have been applied. The optimizations that we applied on the previous algorithm are the following: 1) remove all entries in queue that have score lower than the score of the solution found,

Algorithm 5 Plug-in functions for the MinRisk heuristic.

```
function ChoosePath(node n_s, int budget, node set V, string mode)
    priority_queue q ← ∅                                ▷ queue with path options to explore
    path bp ← null                                      ▷ best path so far
    int bdiff ← −∞                                      ▷ best/max diff so far
    if mode = NORMAL then                               ▷ use normal estimates
        cmode, gmode ← mean, mean
    else                                                ▷ use optimistic estimates
        cmode, gmode ← min, max
    end if
    insert(q, budget, < (n_s, budget), budget >)        ▷ start path
    while q ≠ ∅ do
        p ← removeHighPrio(q)                           ▷ get path option with higest score
        n_i, b ← p.hops[len(p)].n, p.b
        for each n_j | e_ij ∈ E do                      ▷ each neighbor of n_i
            remb ← min(b + g_i^mode, B_max) − c_ij^mode ▷ budget after this hop
            if remb > 0 then                            ▷ budget not exhausted
                diff ← min(remb, p.hops[len(p)].diff)
                if diff > bdiff ∨ diff = bdiff ∧ len(p) < len(bp) then
                    if n_j ∈ V then
                        deleteLowPrio(q, diff)          ▷ delete paths with lower score
                        bdiff, bp ← diff, p ⊕ (n_j, diff)
                    else
                        insert(q, diff, < p ⊕ (n_j, diff), remb >)
                    end if
                end if
            end if
        end for
    end while
    return bp
end function

function UnexpectedOutcome(path p, int k, int diff, string mode)
    if mode = NORMAL then
        return |p.hops[k].diff − diff| / p.hops[k].diff > Threshold_normal
    else
        return |p.hops[k].diff − diff| / p.hops[k].diff < Threshold_optimistic
    end if
end function
```

and 2) do not add to queue nodes that have been visited and the budget of the previous visit was greater or equal.

Chapter 7

Reference algorithms

To put the performance of the above heuristics into perspective, we use three additional algorithms as a reference.

7.1 Hop minimization

As a first reference we use a hop minimization algorithm. This algorithm chooses every time to visit the node of interest that is closest in hops to the node that the vehicle is currently located and travels using the shortest, in hops, path. This approach is implemented as a BFS algrithm. In our case that each cost distribution is the same, this path planning will behave as a constrained shortest path algorithm.

7.2 Ant colony optimization

As a second reference, we use an ant colony optimization technique [13]. This is a probabilistic approach for solving path finding problems, and has been used expensively in previous works, which study vehicle routing problems (VRP) as well as traveling salesman problems (TSP). More specifically, in our case, a number n of ants are generated for m generations and each ant has a specific budget. Each ant follows at first a randomly chosen route. In our case the ants leave the pheromone trail of their path if they visit all of the nodes of interest or their budget is exhausted but they have achieved the best score. The pheromone trail increases the probability

of an ant of a future generation to choose that specific path. When an ant stops traveling, which means that its budget is exahusted or it has visited all of the nodes of interest, it submits its score. The path returned is the one with the maximum submited score

7.3 Oracle

As a third reference, we use an oracle algorithm that has full knowledge of the outcome of any action that will be taken — it knows the actual edge costs and node gains that will apply each time the vehicle would cross and edge and visit a depot node, respectively. We refer to this as the *Oracle*. The algorithm works in three steps. In the first step, its finds the minimum cost path from each $n_i \in V$ to each $n_d \in G$ (based on the actual edge costs). Then, for each depot node n_d, it constructs "out" and "in" node clusters. Node $n_v \in V$ belongs to the out cluster of n_d if it can be reached from n_d with the smallest cost compared to any other depot node. Node $n_v \in V$ belongs to the in cluster of n_d if the cost for reaching n_d is the smallest among all other depot nodes. In the second step, the algorithm visits as many nodes of interest it can based on the current budget, before returning to a depot node to gain some energy (this is done based on the adapted Bellman-Ford code discused earlier). This is repeated, until it is not possible to visit some of the remaining nodes of interest and then return to some depot node. Finally, in a third step, the algorithm tries to visit as many nodes as possible with the remaning budget (without returning to a depot node). This is done by runnning the Held-Karp algorithm [12] (which is an exact solution to the TSP). Initially, the algorithm is run for all remaining nodes of interest, let k. If no solution is found for k, the algorithm is run for $k-1$, and if no solution is found, then it is run for $k-2$ etc. If a solution is found for $k \leq 2$, the path returned by the Held-Karp algorithm is adopted. For $k = 1$, the algorithm simply picks the cheapest path to any node of interest, and marks the node as visited if the budget suffices to reach that node.

Chapter 8

Evaluation

8.1 Setup

We evaluate the above heuristics through a series of experiments. For the graph G, we use a 10×10 grid with a total of 100 nodes. Each node is connected only to its horizontal and vertical neighbours. So, each node that is not at the periphery has 4 edges. The diameter d of the graph is 18 hops.

We fix the number of depot nodes to 2. The depot nodes are chosen randomly once, and remain the same for every experiment. More specifically, the two depot nodes are the green nodes, at position $(2,5)$ and $(8,6)$ in the grid. Also, in all experiments, the vehicle always starts its journey from the same node. This node is chosen under the restriction that the vehicle can reach at least one depot node with the initially available budget. More specifically, the starting node is blue node, at position $(0,0)$ in the grid. Figure 8.1 shows the setup. With no loss of generality, we set the maximum budget limit $B_{max} = 1000$. We also set the initial budget of the vehicle $b = B_{max} = 1000$.

The set of nodes V to be visted by the vehicle are chosen randomly. We experiment with different V_x where $x = \frac{|V_x|}{|N|}$, for $x = 5\%, 10\%, 20\%,\ 100\%$. For each $V_x, x < 100$, we test 3 different (randomly generated) sets, $V_{x_k}, 1 \leq k \leq 3$. To draw more meaningful conclusions across the different V_x, in each case, we ensure that $V_{5_k} \subset V_{10_k} \subset V_{20_k}$.

The gain of depot nodes follows a *symmetrical double-truncated normal distri-*

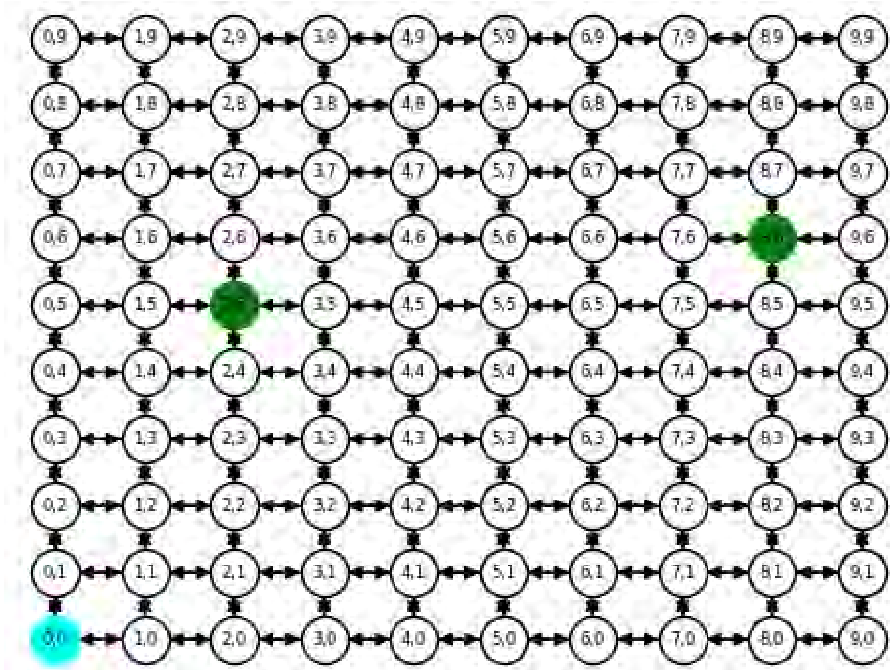


Figure 8.1: Graph used in the experiments. The start node $(0, 0)$ is in blue and the depot nodes $(2, 5), (8, 6)$ in green.

bution with mean value g^{mean}, lower bound $g^{min} = g^{mean} - \frac{g^{mean}}{3}$ and upper bound $g^{max} = g^{mean} + \frac{g^{mean}}{3}$. In all experiments, we use the random distribution for all depot nodes, with $g^{mean} = \frac{3}{4} \times B_{max}$. The rationale behind this is that, when the vehicle reaches a depot with marginally exhausted budget, we want it to continue its travel with expected budget the 75% of the maximum budget.

The edge costs also follow a *symmetrical double-truncated normal distribution* with mean value c^{mean}, lower bound $c^{min} = c^{mean} - \frac{c^{mean}}{2}$ and upper bound $c^{max} = c^{mean} + \frac{c^{mean}}{2}$. However, we perform experiments for different edge cost distributions — in each experiment, all edges follow the same cost distribution. More specifically, we let $c^{mean} = \frac{B_{max}}{a}$, where a is the so-called *autonomy degree*, corresponding to the average number of hops that can be performed by the vehicle with a maximum initial budget. We perform experiments for 4 different autonomy degrees, $\frac{1}{2} \times d = 9$ (low), $\frac{2}{3} \times d = 12$ (low-medium), $\frac{5}{6} \times d = 15$ (medium-high) and $d = 18$ (high). Clearly, higher degrees of autonomy enable the vehicle to perform a larger number of hops and visit more of nodes before having to gain some energy at a depot node.

For each autonomy degree, we produce 100 different edge cost and node gain

scenarios. For each scenario, the cost of each edge and the gain of each depot node is produced offline. These values are stored in a scenario file, from where they are retrieved at runtime. Note that the cost of an edge changes each time the edge is crossed; in the scenario file, 5 different values are produced for each edge, which are retrieved in a round-robin fashion each time the vehicle crosses that particular edge. The same applies to the gain for each depot node. The edge cost and node gain values of each scenario are a priori known only to the oracle heuristic.

For each configuration (combination of a and Vx), we perform 3×100 runs (100 scenarios for each V_{x_k}), and present the average *coverage* and *efficiency* scores achieved by each heuristic. In order for the ant colony optimization method (ACO) to produce good results, in each run we use 10 ants and 200 generations of ants (recall that the method returns the path with the best score achieved by any ant).

Finally, the $Thresholds$ used in the *UnexpectedOutcome*() function is set as the best result of the first series of experiments. We will discuss about this experiment in the next subsection.

8.2 Testing different thresholds

In these experiments we vary the thresholds used in the *UnexpectedOutcome*(). Based on the different autonomy degrees, the number of destination nodes and the heuristics used, we evaluate different percentages and decide what is the best percentage to be used as the threshold. We compare 3 threshold values for each mode. The values of the thresholds tested are 10%, 25%, 40% and 4%, 10%, 16%, for the normal and optimistic mode, respectively. The fact that we use a more strict observation for the optimistic mode, is due to the "unrealistic" assumption made by the mode. The best case scenario, which is what the optimistic mode assumes, is hard to be achieved, so we need to lower the toleration for errors. In this experiment, we calculate the average score and average number of plannings (how many times does a planning for a new path is needed). The average values are computed on each autonomy degree from all the different V_x and scenarios.

In the Figures-8.2,8.3,8.4,8.5, we can see the results of the current experiment.

The first observation we make is that for the different values of the $Threshold_{optimistic}$ we see no differences at all except one small deviation on the number of plannings for the 2 heuristics. That means that when the mode changes to optimistic, the UV travels pointlessly. As for the $Threshold_{normal}$, the different values affect the 2 models more obviously. For the maximizing budget heuristic, we have a rise on the number of plannings when the threshold takes small values, but we also have a rise in the score. So we choose to use for this model a low value for threshold. However for the minimizing risk, when decreasing the threshold we have a rise only in the number of plannings and not in the score. So in this model it is obvious that it is in our interest to keep a high value as threshold.

We come to conlusion that the best value for $Threshold_{optimistic}$ is 4% and the best value for $Threshold_{normal}$ is 10% for the maximizing budget heuristic and 40% for the minimizing risk heuristic.

8.3 Comparison with MinHops and ACO

For this set of experiments we use the best thresholds that we found from the above experiment. For each V_x we calculate the average score of the 2 models over the 4 autonomy degrees. As a mean of comparison we use the ACO algorithm and the minimizing hops algorithm that we previously mentioned.

In Figure-8.6 we can see the results of the current series of experiments. It's clear to see that the maximizing budget heuristic has the best score over all the other algorithms in any configuration and any V_x. The score of our other heuristic, on the other hand, is low. It deviates from the maximizing budget algorithm more and more, with the raise of the number of nodes of interest and when lowering the autonomy degree. It is intreseting, but also rational, the fact that the minimizing risk heuristic has the same score as the minimizing hops algorithm in the special case of V_{100} (Figure-8.6e). This is because risk the minimization heuristic will always

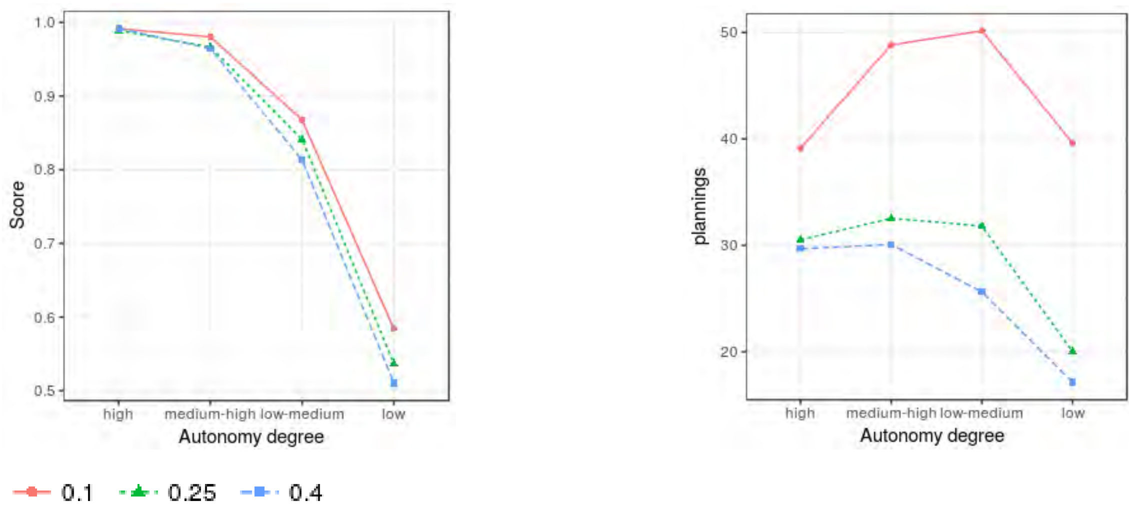


Figure 8.2: Effect of $Threshold_{normal}$ on the MaxBudget heuristic.

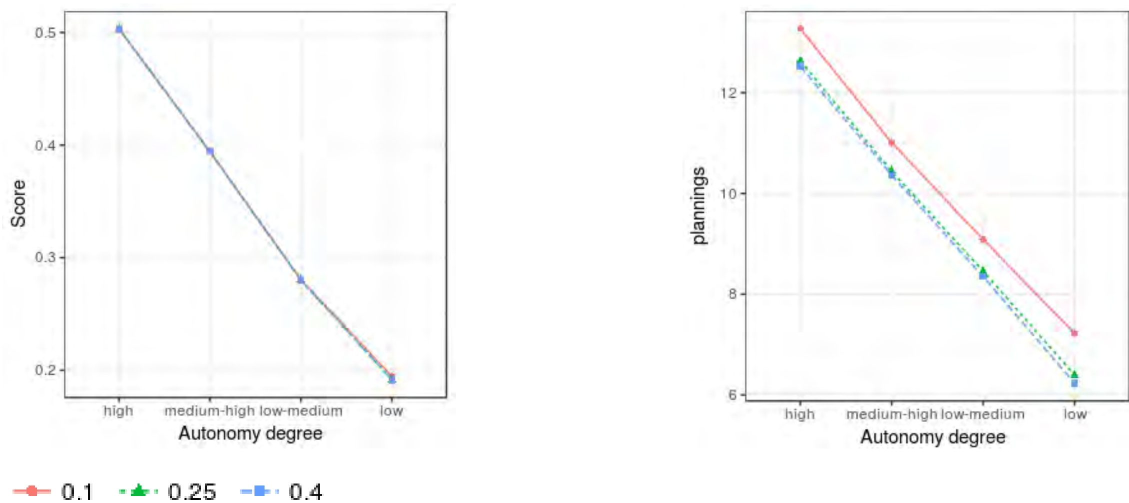


Figure 8.3: Effect of $Threshold_{normal}$ on the MinRisk heuristic.

pick as next destination, the node that has maximum minimum remaining budget. So when there is always a node of interest in the 1 hop neighborhood, 2 hop paths, or longer, will never be examined. It is also intresting the fact that the ACO seems to have a little better score in the normal configuration than the wide one. It could mean that the topology has high enough consumption that the ants tend to the depot nodes earlier but not high enough to stop them from traveling.

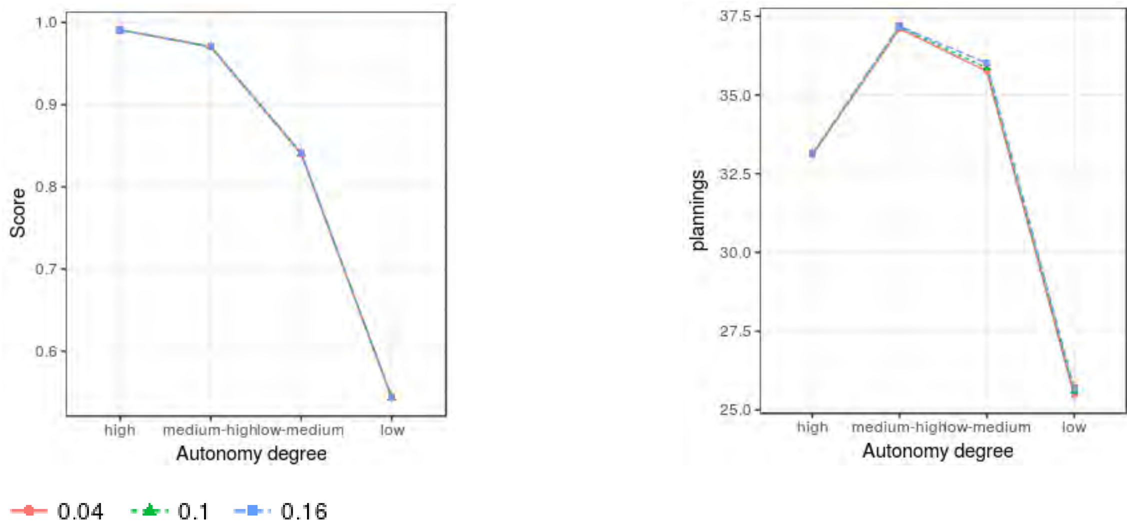


Figure 8.4: Effect of $Threshold_{optimistic}$ on the MaxBudget heuristic.

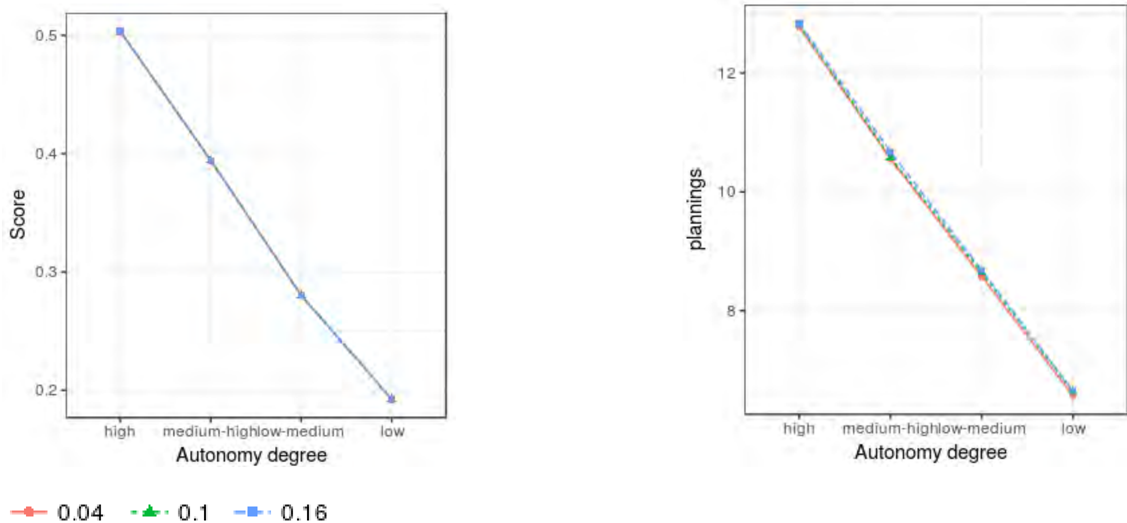
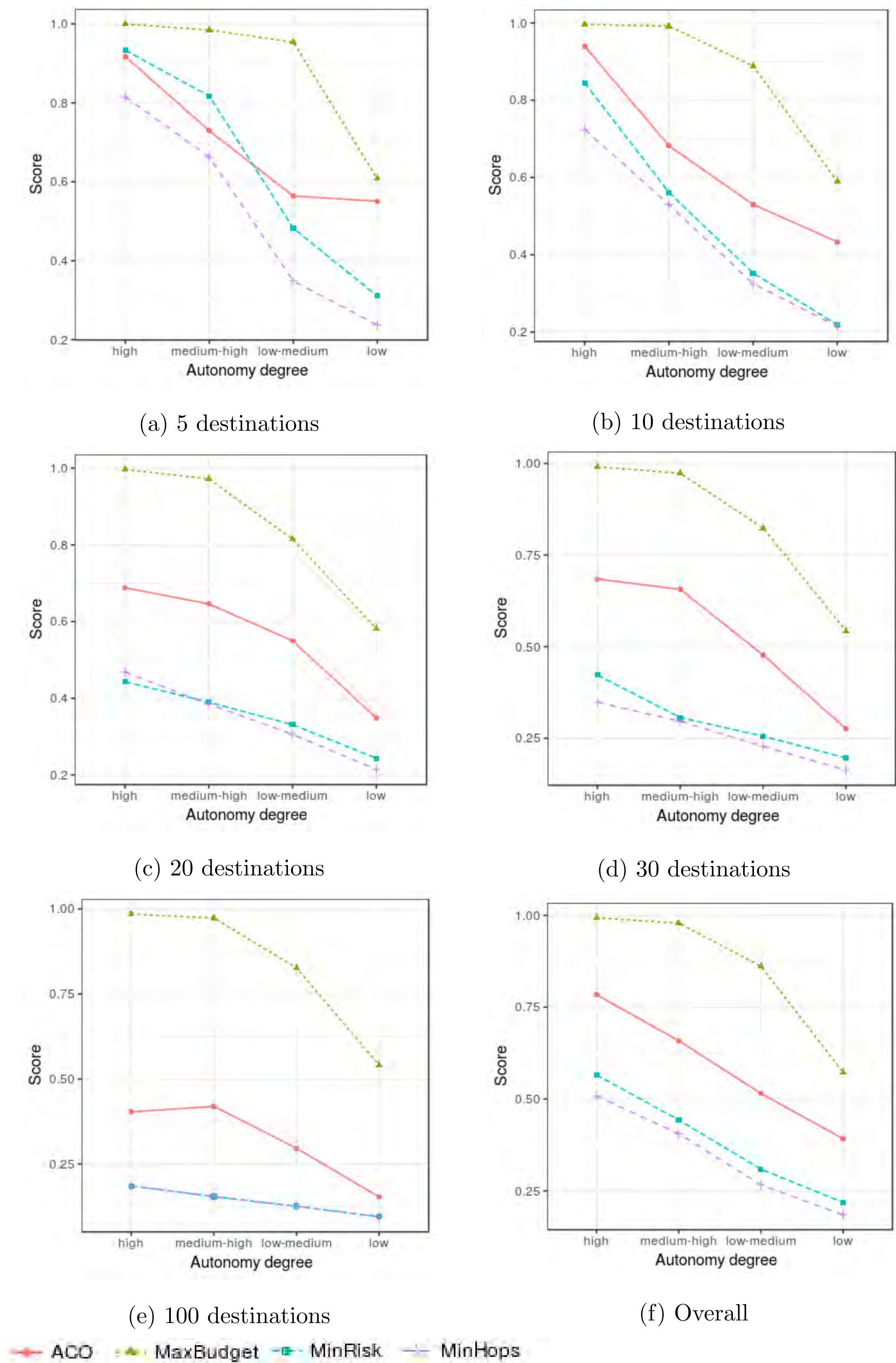


Figure 8.5: Effect of $Threshold_{optimistic}$ on the MinRisk heuristic.

8.4 Comparison with oracle

In the final set of experiments we compare our two heuristic solutions with the oracle we mentioned before. In order not to add an addition complexity to the search of an exact solution in the oracle algorithm, just for this experiment we use only one value from each scenario (we do not pick the values with round-robin order, we use only one).

In Figure 8.7 we can see the results of this experiment. Now it is more clear how much, each heuristic, deviated from the best score. We can see that for high



(a) 5 destinations

(b) 10 destinations

(c) 20 destinations

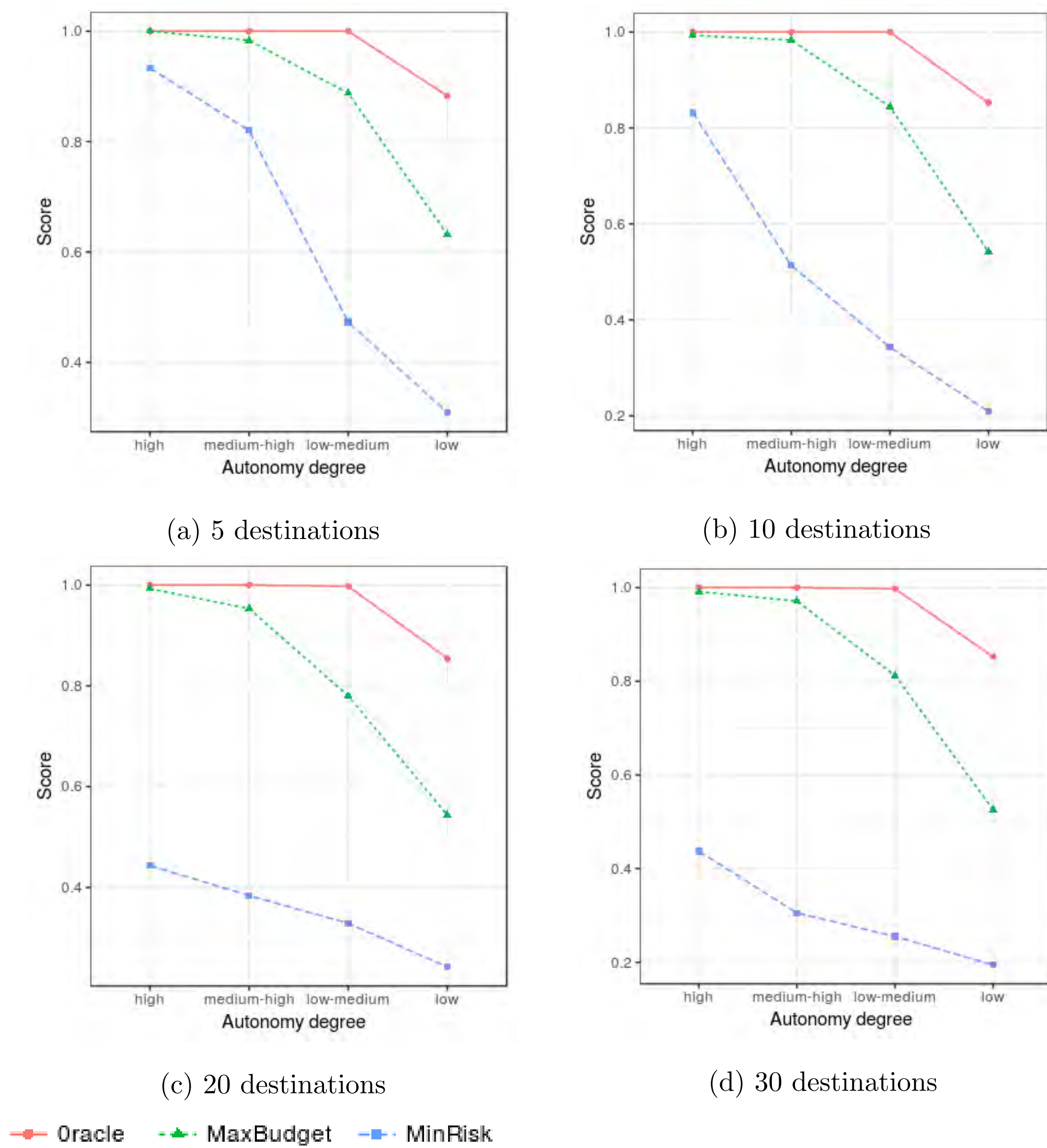
(d) 30 destinations

(e) 100 destinations

(f) Overall

Figure 8.6: Comparison of our heuristics with ACO and minimizing hops algorithm.

autonomy degrees the maximizing budget heuristic has a score very close to the best score. When the autonomy degree is lower then both of the heuristics deviate more from the best score. We can also observe that, maximizing budget heuristic has better score when the nodes of interest increase in number. This is because the heuristic will choose to visit the nodes that are closer to the depot nodes first (because these nodes are the nodes will have higher remaining budget) and then visit the ones that is more likely to fail. So it is more possible to visit a higher percentage of the nodes of interest.



(a) 5 destinations

(b) 10 destinations

(c) 20 destinations

(d) 30 destinations

Figure 8.7: Comparison of our heuristics and the oracle.

Chapter 9

Conclusion

In this project, we created and tested two heuristic algorithms, with target to maximize the number of the nodes of interest that they visit before running out of budget, while traveling in a stochastic environment. The design of our first heuristic, maximizing budget, relies on energy efficient paths, while our second heuristic, minimizing risk, relies on paths that are not close to infeasibility.

As we show the maximizing budget heuristic achieves better scores in all the autonomy degrees and all the different sizes of sets of nodes of interest. This is because this approach is more farsighted. Since we have multiple destination nodes and not a single one, when we pick paths that maximize the budget, is much more efficient, because we also take into account the future paths as well. If we had a single target, then minimizing the risk of the paths could prove that it is a better approach.

As future work there are two intresting things to be done. First of all, further experimenting. The above experiments gave us an idea of the goodness of each algorithm, but retrieving real data would be even more helpful to understand the performance of each algorithm. As real data we refer to the consumption of a vehicle in an unstable environment, to be applied as cost distributions. But also real world gain values to be applied into the gain distribution. Secondly, we could find more heuristic algorithms. Now that we have a first image of the results, it would be a great support to design new algorithms. For example we need farsighted algorithms that minimize the risk of each travel, so a combination of the two proposed heuristic

would have great results (using adapted bellman ford to calculate paths but pick the one with the maximum minimum remaining budget).

Bibliography

[1] Federico Alonso, M Jesús Alvarez, and John E Beasley. A tabu search algorithm for the periodic vehicle routing problem with multiple vehicle trips and accessibility restrictions. *Journal of the Operational Research Society*, 59(7):963–976, 2008.

[2] Enrico Angelelli and Maria Grazia Speranza. The periodic vehicle routing problem with intermediate facilities. *European journal of Operational research*, 137(2):233–247, 2002.

[3] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The optimal routing problem in the context of battery-powered electric vehicles. In *CPAIOR Workshop on Constraint Reasoning and Optimization for Computational Sustainability (CROCS)*, 2010.

[4] Russell W Bent and Pascal Van Hentenryck. Scenario-based planning for partially dynamic vehicle routing with stochastic customers. *Operations Research*, 52(6):977–987, 2004.

[5] Dimitris J Bertsimas. A vehicle routing problem with stochastic demand. *Operations Research*, 40(3):574–585, 1992.

[6] Jean-François Cordeau, Michel Gendreau, and Gilbert Laporte. A tabu search heuristic for periodic and multi-depot vehicle routing problems. *Networks: An International Journal*, 30(2):105–119, 1997.

[7] Alan L Erera, Juan C Morales, and Martin Savelsbergh. The vehicle routing problem with stochastic demand and duration constraints. *Transportation Science*, 44(4):474–492, 2010.

[8] Manlio Gaudioso and Giuseppe Paletta. A heuristic for the periodic vehicle routing problem. *Transportation Science*, 26(2):86–92, 1992.

[9] Michel Gendreau, Gilbert Laporte, and René Séguin. An exact algorithm for the vehicle routing problem with stochastic demands and customers. *Transportation science*, 29(2):143–155, 1995.

[10] Michel Gendreau, Gilbert Laporte, and René Séguin. A tabu search heuristic for the vehicle routing problem with stochastic demands and customers. *Operations Research*, 44(3):469–477, 1996.

[11] Maher Helaoui. Extended shortest path problem - generalized dijkstra-moore and bellman-ford algorithms. In *Proceedings of the 6th International Conference on Operations Research and Enterprise Systems - Volume 1: ICORES,*, pages 306–313. INSTICC, SciTePress, 2017.

[12] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[13] Karl O Jones. Ant colony optimization, by marco dorgio and thomas stützle, a bradford book, the mit press, 2004, xiii+ 305 pp. with index, isbn: 0-262-04219-3, 475 references at the end.(hardback£ 25.95)-. *Robotica*, 23(6):815–815, 2005.

[14] Astrid S Kenyon and David P Morton. Stochastic vehicle routing with random travel times. *Transportation Science*, 37(1):69–82, 2003.

[15] Gilbert Laporte, Francois Louveaux, and Hélène Mercure. The vehicle routing problem with stochastic travel times. *Transportation science*, 26(3):161–170, 1992.

[16] Gilbert Laporte, François V Louveaux, and Luc Van Hamme. An integer l-shaped algorithm for the capacitated vehicle routing problem with stochastic demands. *Operations Research*, 50(3):415–423, 2002.

[17] Jonathan Las Fargeas, Baro Hyun, Pierre Kabamba, and Anouck Girard. Persistent visitation with fuel constraints. *Procedia-Social and Behavioral Sciences*, 54:1037–1046, 2012.

[18] Vera Mersheeva. *UAV Routing Problem for Area Monitoring in a Disaster Situation*. PhD thesis, 2015.

[19] Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. Efficient energy-optimal routing for electric vehicles. In *AAAI*, pages 1402–1407, 2011.

[20] Nicola Secomandi. A rollout policy for the vehicle routing problem with stochastic demands. *Operations Research*, 49(5):796–802, 2001.

[21] Duygu Taş, Nico Dellaert, Tom Van Woensel, and Ton De Kok. Vehicle routing problem with stochastic travel times including soft time windows and service costs. *Computers & Operations Research*, 40(1):214–224, 2013.

[22] T Van Woensel, L Kerbache, H Peremans, and Nico Vandaele. A vehicle routing problem with stochastic travel times. In *Fourth Aegean International Conference on Analysis of Manufacturing Systems location, Samos, Greece*, 2003.

[23] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012.