Tackling Multi-Agent Routing in an Orienteering Problem Setting

Diploma Thesis

Author :
Stergios Plataniotis

Committee :
Associate Professor Georgios Chalkiadakis (Supervisor)
Associate Professor Michail Lagoudakis
Professor Antonios Deligiannakis

Thesis submitted as partial fulfillment for the degree of Diploma in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Technical University of Crete
Chania
April 2021

Abstract

The Orienteering Problem is a combinatorial optimization problem which constitutes a generalization of the Travelling Salesman Problem. It can be presented as a graph, in which each node is associated with a reward, while each edge is associated with a cost. With the starting and ending nodes fixed, one has to find a path that maximizes the cumulative reward (or "score"), while maintaining a budget. There may also be more limitations, such as an extra cost of visiting each node or knapsack constraints. Such problems are usually solved via heuristics because of their NP-hard complexity. To this end, we extend this competitive setting to a multi-agent routing problem with the addition of congestion-related discounts, and take advantage of Artificial Intelligence methods to address it. Specifically, we model our extended problem in two different ways—i.e., as a multi-agent Markov Decision Process (MDP), and as Partially Observable MDP (POMDP); and employ multi-agent Reinforcement Learning (MARL) and Partially Observable Monte Carlo Planning (POMCP), respectively, to find good solutions. Our MARL solution employs a Coordination Graph communication format and the Sparse Cooperative Q-learning algorithm. For our POMCP algorithm, we model congestion as uncertainty countered by belief-particle filtering. Overall, we put forward six different algorithmic variants to tackle this problem, and provide an analysis of their performance via experimental simulations.

Περίληψη

Το Πρόβλημα του Προσανατολισμού είναι ένα πρόβλημα συνδυαστικής βελτιστοποίησης, και αποτελεί γενίκευση του προβλήματος του πλανώδιου πωλητή. Μπορεί να αναπαρασταθεί σαν πρόβλημα εύρεσης μονοπατιού πάνω σε έναν γράφο, στον οποίο κάθε κόμβος συνδέεται με μία αμοιβή, ενώ η διάσχιση κάποιας ακμής με κάποιο κόστος. Γνωρίζοντας τον αρχικό και τον τελικό κόμβο, το ζητούμενο είναι η εύρεση ενός μονοπατιού που να τα συνδέει το οποίο μεγιστοποιεί τις συνολικές απολαβές (το "σκορ"), χωρίς την υπέρβαση ενός αρχικού προϋπολογισμού. Μπορεί να υπάρχουν και επιπλέον περιορισμοί, όπως κάποιο περαιτέρω κόστος για την επίσκεψη σε κάθε κόμβο, ή περιορισμοί σακιδίου. Καθώς το πρόβλημα είναι NP-hard, οι διάφορες παραλλαγές του αντιμετωπίζονται συνήθως με χρήση προσαρμοσμένων σε αυτές ευρετικές μεθόδους. Στην παρούσα εργασία, επεκτείνουμε αυτό το μοντέλο μετατρέποντάς το σε ένα πολυπρακτορικό πρόβλημα εύρεσης μονοπατιών, με την προσθήκη μιας "έκπτωσης αξίας" στη σχετιζόμενη με κάθε κόμβο αμοιβή, ανάλογα με τη συμφόρηση του εν λόγω κόμβου. Κατόπιν, αντιμετωπίζουμε το νέο αυτό πρόβλημα εφαρμόζοντας μεθόδους Τεχνητής Νοημοσύνης. Συγκεκριμένα, μοντελοποιούμε το πρόβλημα ως πολυπρακτορική Διαδικασία Αποφάσεων Markov καθώς και ως Μερικώς Παρατηρήσιμη Διαδικασία Αποφάσεων Markov, και το αντιμετωπίζουμε με τη χρήση μεθόδων πολυπρακτορικής ενισχυτικής μάθησης (multiagent reinforcement learning - MARL) και σχεδιασμού Monte-Carlo (με τον αλγόριθμο Partially Observable Monte Carlo Planning - POMCP) αντίστοιχα. Οι μέθοδοι MARL που χρησιμοποιούμε αξιοποιούν τον αλγόριθμο Sparse Cooperative Q-learning πάνω σε Συνεργατικούς Γράφους. Για τη λειτουργία του POMCP αλγορίθμου μας, μοντελοποιούμε τη συμφόρηση σε κάθε κόμβο ως αβεβαιότητα, και την αντιμετωπίζουμε με "φιλτράρισμα σωματιδίων". Συνολικά προτείνουμε έξι διαφορετικές αλγοριθμικές τεχνικές για την αντιμετώπιση του προβλήματος, και αξιολογούμε την απόδοσή τους πειραματικά με χρήση κατάλληλων προσομοιώσεων.

Acknowledgements

First, I would like to thank my supervisor, Associate Prof. Georgios Chalkiadakis, for his guidance throughout this work. I would also like to thank Dimitrios Troullinos, PhD student at Technical University of Crete, for his contribution with useful comments and adjustments.

Last but definitely not least, I would like to express my gratitude to my family, as well as Emmanouela, for their unconditional support and encouragement all these years.

I dedicate this thesis to my late grandmother, Anastasia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence is a vast subfield of Computer Science. It refers to the ability of machines to solve tasks that typically show of a level of intelligence, and its applications can be found in a plethora of problems. More notably, AI algorithms are used to solve problems using natural language processing, planning, knowledge representation, artificial neural networks and more. Those have led to multiple breakthroughs with autonomous driving, spam mail detection, robots, speech recognition and personalized advertising, to name a few. The long-term goal of AI is Artificial General Intelligence (AGI), which aspires to allow an autonomous entity (a rational autonomous agent) that possesses it to perform any task a human can do. On the contrary, most AI systems nowadays are designed to solve only a specific problem or a family of problems, and thus correspond to the so-called narrow AI.

Usually, in some AI applications, we use one agent that acts in a specified, frequently simulated, environment. But, more often than not, a problem requires the introduction of more than one agents. This is true for problems in which the relationship and the interactions between the agents are essential to the problem itself, or in problems where a decentralized approach is indispensable. The systems that feature multiple agents are called Multi-Agent Systems (MAS), and have found application in patrolling and disaster response among others.

A subfield of AI that has been used to tackle multiple problems is Reinforcement Learning (RL) [Kaelbling et al., 1996]. RL allows an agent to learn in an environment by repeatedly acting in it. In more detail, the agent observes the representation of the current environmental state, takes an action, possibly obtains a reward, observes the current state again and so on. The agent's goal is to ultimately learn a policy (mapping of states to actions), that will maximize its cumulative reward. RL is often paired with the framework of Markov Decision Processes [Kaelbling et al., 1998], which give a more detailed and rigorous portrayal of the environment and its dynamics. Another family of algorithms used to tackle problems is Monte Carlo methods. A popular one is Monte Carlo Tree Search (MCTS) [Coulom, 2006], which uses a tree representation of the environment and utilizes heuristics and randomness as well as RL principles in order to find a sufficient solution.

The Orienteering Problem (OP) is a routing problem formally introduced first with this name by [Golden et al., 1987], but introduced first by [Tsiligirides, 1984] as the Generalized Traveling Salesman Problem (GTSP). It got its name from a group of sports, named Orienteering [Orienteering, 2020], that requires navigating to control points on an uneven terrain with the help of a compass and a special map. Regarding the difference between TSP and the OP, in the former we search for a sequence of nodes that minimizes the total travel cost while having to visit all nodes, whilst in the latter each node is affiliated with a score and we ought to maximize the total score by visiting only a subset of nodes. The feasibility of a path for a given OP instance is determined by a starting time budget. As visiting a node requires a travel time as a cost, the total of travel costs of a path must not exceed the starting budget. More specifically, the nodes are points on the 2-dimensional space and the travel time between two points equals the Euclidean distance between them. A possible obstacle in this setting is that many points can have a large score, though demand a large amount of resources. Thus, a point with a large score is not always a good move.

The need to display some problems with more restrictions as OP raised the need for more OP variations [Fomin and Lingas, 2002; Pěnička et al., 2019; Souffriau et al., 2013] that arise by just presenting more limitations. Some extra limitations can be adding an extra constant travel cost to each node as duration, or an extra budget along with the affiliation of each node with a cost, or by requiring more than one paths. Some popular variants are presented in Chapter 2.

OP finds application in a number of real world problems, some of them being the tourist trip design problem [Gavalas et al., 2015], delivery problem [Archetti et al., 2014], and monitoring [Yu et al., 2016]. There have been some exact algorithms in the literature used to tackle OP, two of them being branch-and-bound and branch-and-cut algorithms, but most of them rely on heuristics, mainly since the OP is NP-hard and an approximative approach is needed. Some algorithms made to tackle OP variations have also used search algorithms like Variable Neighborhood Search (VNS) [Sevkli and Sevilgen, 2006] and Iterated Local Search (ILS) [Gunawan et al., 2015], in addition to Ant Colony Optimization (ACO) [Ke et al., 2008].

1.1 Contribution

Our contribution in this work is the study of multi-agent routing in an OP setting with the addition of congestion at each node. This is interesting and important for mainly two reasons. First, the coexistence of many agents can lead to the finding of multiple good paths, as is the case in the Team Orienteering Problem [Chao et al., 1996], where the computation of multiple paths is requested instead of just one. As for congestion, it can be used to describe real life problems in which congestion opposes efficiency, more accurately, as the patrolling of an area by multiple agents, traffic flow optimization [Verbeeck et al., 2016; Walraven et al., 2016] or the tourist trip design problem [Gavalas et al., 2014; Gavalas et al., 2015]. Hence this problem is interesting to study even though it can be safely considered difficult, since the OP is proven [Golden et al., 1987] to be difficult (NP-hard) and our extension only adds in complexity.

To go into some detail, each agent will have to formulate a path (subset of nodes) in order to maximize its cumulative score while, at the same time, avoiding congested nodes, which brings a penalty in score of size similar to the level of congestion. Our goal is to maximize the summation of the total discounted scores of each agent. By discounted scores we refer to the proportion of the original score an agent receives by visiting a node. The specific proportion is calculated each time given the exact congestion (agents visiting at the same timestep) at the specific node. We must emphasize that, in our setting, all agents move at the same time in a synchronous manner. To be more specific, all agents start from the same starting node and aim to the same ending node, at each timestep each agent chooses the next node it will transit to, all agents travel to their respective desired nodes and after that, they receive the destination's (possibly) discounted score as a reward. This process repeats until all agents have reached the goal node. Typically, the agents will frequently face the dilemma of traveling to an expectedly congested node, versus traveling to a node that would normally lead to a smaller total score, if it was not for congestion.

To tackle this problem, we use AI methods, in the sense of making the agents learn better policies in order to find a good solution. In particular, we formulate the problem as a Partially Observable Markov Decision Process (POMDP) [Kaelbling et al., 1998] and as a multi-agent Markov Decision Process and employ different algorithms to solve it. In the POMDP formulation, each agent acts on its own and models the potential congestion of each node as uncertainty, whereas in the multi-agent MDP formulation we use the framework of Coordination Graphs (CG) [Kok and Vlassis, 2006] that enables the agents to collaborate in order to avoid congested nodes. In the end we utilize six algorithms and measure their performance. More specifically, two algorithms emerge from SparseQ [Kok and Vlassis, 2006], by using two different update methods; two from POMCP [Silver and Veness, 2010], by using the standard version and one exploiting domain-specific knowledge; and also Q-learning and a naive method choosing vertices at random.

1.2 Overview

We now give an overview of the rest of the thesis. In Chapter 2, we define the OP and some of its variants in detail, brief-review state-of-the-art work that has been done to solve OP problems, and present in detail the algorithms SparseQ, POMCP and Q-learning, as these are the algorithms of choice in this thesis to tackle our problem. In Chapter 3, we model our problem as a POMDP and as a multi-agent MDP and describe some problem-specific intricacies. In Chapter 4, we present the performance of our algorithms, tested on six different datafiles of three different variations of the OP, along with box plots depicting the variance of cumulative rewards between the agents, as well as figures that depict the learning process, where needed, and discuss the results. Finally, in Chapter 5, we wrap up our results and contributions, and explain how our approach can be extended in the future.

Chapter 2

Background and Related Work

In this chapter we present the necessary background for this thesis, and also discuss related work.

2.1 The Orienteering Problem

The OP [Golden et al., 1987] is a routing problem which can be seen as a combination of the Knapsack problem and the Travelling Salesman Problem (TSP) [Hoffman et al., 2013]. Given a set of nodes, a time budget and a starting and ending node, its goal is to visit a subset of nodes that maximize the total collected score. The time needed to visit node j from a node i is fixed and, in most cases, symmetric. More than often the set of nodes are basically points on the Euclidean space and thus the Euclidean distance between two points is considered as the travel time. In case that the starting and ending nodes coincide, a tour must be formulated instead of a path.

In summary we have a set on N nodes, where each node i is associated with a score S_i and t_{ij} is the travel time from node i to a node j. If, the travel times are symmetric then $t_{ij} = t_{ji}$.

From a different point of view, it can also be seen as a complete and undirected graph G=(V,A), where each vertex v is associated with a non-negative score and each arc a is associated with a non-negative cost (travel time).

As OP's resemblance to the TSP suggests, OP is proven to be NP-hard [Golden et al., 1987]. That means it is a very hard problem and it is expected that no algorithm can be designed to find the best possible path in polynomial time. A difficulty worth noting, is that a node associated with a relatively high score, can be very expensive to travel to and thus a bad choice.

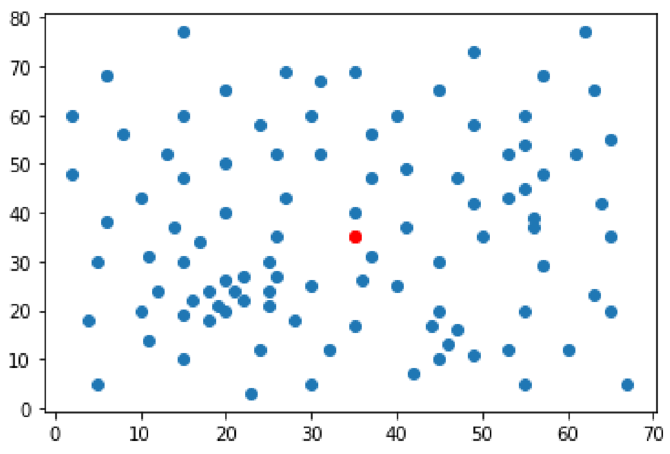
Besides the classic-vanilla setting described above, there are many extensions which have been studied throughout the years.

Some of them are:

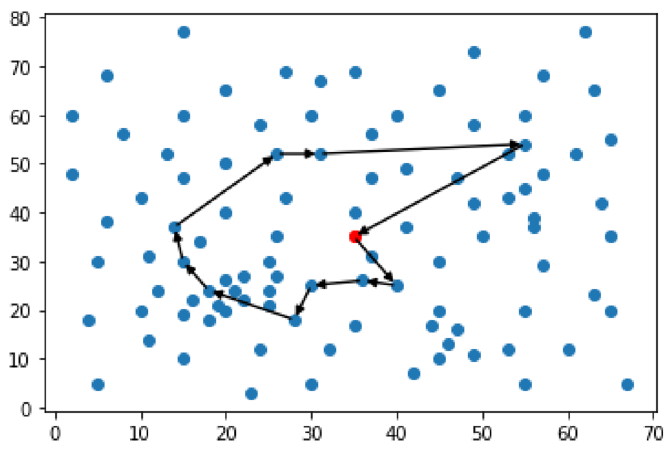
- *Team Orienteering Problem (TOP)* [Chao et al., 1996], where a number of tours to be found is given additionally and thus more than one paths must be found
- *Time Dependent Orienteering Problem (TDOP)* [Fomin and Lingas, 2002], where the travel time between two nodes is neither fixed nor symmetric, and depends on the departure time

- *Orienteering Problem with Time Windows (OPTW)* [Kantor and Rosenwein, 1992], where each node has a fixed time window (opening and closing time) in which it operates. Visiting a node before its opening leads to waiting time, while visiting a node after its closing is infeasible. On top of that, each node has an extra time cost called duration.
- *Multi-Constraint Team Orienteering Problem with Multiple Time Windows (MCTOPMTW)* [Souffriau et al., 2013], where a predetermined budget constraint is added along with budget costs for each node and each node has multiple time windows instead of just one. In addition, a knapsack constraint separates the nodes in types, and limits the number of nodes we can visit for each type.

In Figure 2.1 we can see an example of an MCTOPMTW instance. Figure 2.1a displays the vertices of the file as points on Euclidean space while Figure 2.1b depicts an example path. In this specific instance the starting and ending nodes coincide and can be perceived as the red point.



(a) MCTOPMTW instance file points



(b) example solution

Figure 2.1: OP instance on Euclidean space

2.1.1 Mathematical Formulation

In this subsection, we provide a formulation of OP as an integer programming model. Since we cannot possibly provide the formulation for each OP variant, we will only consider OP and discuss what to be added, mainly for the variants that will be used for testing in the next chapters.

For convenience, we will determine the following decision variables : $x_{ij} = 1$ if we visit node j after node i and 0 otherwise, and u_i to determine the position of visited nodes in the path. Additionally, we assume N nodes, where each has a score S_i with $i = 1, \ldots, N$. If the starting node is also the goal node of the instance, we may add a node N identical to the starting node 1. With respect to that we have the following rules:

1. $\max \sum_{i=2}^{N-1} \sum_{j=2}^{N} S_i x_{ij}$
2. $\sum_{j=2}^{N} x_{1j} = \sum_{i=1}^{N-1} x_{iN} = 1$
3. $\sum_{i=1}^{N-1} x_{ik} = \sum_{j=2}^{N} x_{kj} \leq 1, \forall k = 2, \ldots, (N-1)$
4. $\sum_{i=1}^{N-1} \sum_{j=2}^{N} t_{ij} x_{ij} \leq T_{max}$
5. $2 \leq u_i \leq N, \forall i = 2, \ldots, N$
6. $u_i - u_j + 1 \leq (N-1)(1 - x_{ij}), \forall i, j = 2, \ldots, N$

Expression (1) ensures that the total score is maximized, constraint (2) that path starts from node 1 and ends at node N, constraint (3) that each node is visited at most once, constraint (4) that the total time needed to traverse the path doesn't exceed T_{max}, and at last constraints (5) and (6) prevent subtours [Miller et al., 1960]. Subtours are essentially cycles in the graph. In our case, the solution must be a path or, if nodes 1 and N coincide, a tour. In any of those cases the solution cannot contain subtours.

Since we will use both TOP and MCTOPMTW instances during the experimental evaluation, it is deemed necessary to provide the mathematical formulations for these two, as well. For the TOP we need to guarantee that the above constraints apply for each tour while, in similar fashion, for the MCTOPMTW we need to make sure that the budget is not exceeded, the knapsack constraints are met, and the nodes are visited during their time window.

For the TOP format we use the decision variables: x_{ijp}, y_{ip} and u_{ip}. Where x_{ijp} is 1 if in path p a visit in node i is followed by a visit in node j and 0 otherwise, y_{ip} if in path p node i is visited and 0 otherwise, and finally u_{ip} equals the position of node i in path p. In total we need to compute P paths. With that said we have the following rules:

1. $max \sum_{p=1}^{P} \sum_{i=2}^{N-1} S_i y_{ip}$
2. $\sum_{p=1}^{P} \sum_{j=2}^{N} x_{1jp} = \sum_{p=1}^{P} \sum_{i=1}^{N-1} x_{iNp} = P$
3. $\sum_{p=1}^{P} y_{kp} \leq 1, \forall k = 2, \ldots, n-1$
4. $\sum_{i=1}^{N-1} x_{ikp} = \sum_{j=2}^{N} x_{kjp}, \forall k = 2, \ldots, N-1, \forall p = 1, \ldots, P$

5. $\sum_{i=1}^{N-1} \sum_{j=2}^{N} t_{ij} x_{ijp} \leq T_{max}, \forall p = 1, \ldots, P$
6. $2 \leq u_{ip} \leq N, \forall i = 2, \ldots, N, \forall p = 1, \ldots, P$
7. $u_{ip} - u_{jp} + 1 \leq (N-1)(1 - x_{ijp}), \forall i, j = 2, \ldots, N, \forall p = 1, \ldots, P$

Where, expression (1) displays the goal which is to maximize the total score, constraint (2) ensures that each path has to start in the starting node 1, and end in the ending node N, constraint (3) ensures that each node cannot be visited 2 times or more, equation (4) makes certain that the path is connected according to the order of the nodes, constraint (5) ensures that the total time spent for each path does not surpass the given time budget T_{max}, constraints (6) and (7) are needed to prevent subtours.

Similarly, for the MCTOPMTW format the variables x_{ijm}, y_{iwm}, s_{im}, O_{icm}, C_{iwm}, e_{imz}, E_z, L are employed to make the mathematical rules clearer. In detail, x_{ijm} is 1 if in tour m a visit in node i is followed by a visit in node j and 0 otherwise, y_{iwm} is 1 if node i is visited during time window w in tour m and 0 otherwise, s_{im} is the start of the visit at node i in tour m, O_{icm} and C_{iwm} are the opening and closing times of time window w of node i in tour m, e_{imz} is the corresponding cost regarding the knapsack constraint z for node i in tour m, E_z is the budget cost for the knapsack constraint z, and, at last, L is a large constant. The rules are:

1. $max \sum_{m=1}^{M} \sum_{w=1}^{W} \sum_{i=2}^{N-1} S_i y_{iwm}$
2. $\sum_{m=1}^{M} \sum_{j=2}^{N} x_{1jm} = \sum_{m=1}^{M} \sum_{i=1}^{N-1} x_{inm} = M$
3. $\sum_{i=1}^{N-1} x_{ikm} = \sum_{j=2}^{N} x_{kjm} = \sum_{w=1}^{W} y_{kwm}, \forall k = 2, \ldots, N-1, \forall m = 1, \ldots, M$
4. $s_{im} + t_{ij} - s_{jm} \leq L(1 - x_{ijm}), \forall i, j = 1, \ldots, N, \forall m = 1, \ldots, M$
5. $\sum_{m=1}^{M} \sum_{w=1}^{W} y_{iwm} \leq 1, \forall i = 1, \ldots, N$
6. $\sum_{m=1}^{M} \sum_{w=1}^{W} \sum_{i=1}^{N} e_{imz} y_{iwm} \leq E_z, \forall z = 1, \ldots, Z$
7. $\exists w \in 1, \ldots, W$ such that $O_{iwm} \leq s_{im} \leq C_{iwm}, \forall i = 1, \ldots, N, \forall m = 1, \ldots, M$

As previously stated, equation (1) sets up the goal of maximizing the total score, constraint (2) ensures that the starting and ending nodes are 1 and N respectively, constraints (3) and (4) guarantee the connectivity of each path, rule (5) ensures that a node cannot be visited more than once, constraint (6) are the knapsack constraints introduced in this variation, constraint (7) obliges the start of the visit of each node in respect of its time windows.

Most OP variations' mathematical formulations can be found in the survey by [Vansteenwegen et al., 2011]. The formulation specifically for MCTOPMTW can be found in [Souffriau et al., 2013].

2.1.2 Applications and Solution Approaches

OP and its modifications have been applied and used to model and study a plethora of problems of similar fashion. Some of them being the Tourist Trip Design Problem, service scheduling, surveillance activities, delivery scheduling and more.

Due to the problem being NP-hard [Golden et al., 1987], as it is expected from its resemblance to the TSP, it seems more natural for it to be tackled with heuristic algorithms. Despite that, exact algorithms have also been tested with success mainly on smaller instances. Some state-of-the-art work uses iterated local search, tabu search, ant-colony optimization and more [Gunawan et al., 2016].

2.2 Markov Decision Processes

A Markov Decision Process (MDP) [Kaelbling et al., 1996; Sutton and Barto, 2018] is a mathematical framework used for modelling decision making problems and forms an extension to Markov Chains. By definition, the MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ where:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{T}(s, \mathrm{a}, s') = \mathcal{P}_r(s'|s, \mathrm{a})$ is the transition function
- $\mathcal{R}(s, \mathrm{a})$ is the reward function
- γ is the discount factor

We assume that we have full observability over the MDP. The agent lies in a state $s \in \mathcal{S}$, decides to take an action $\mathrm{a} \in \mathcal{A}$, results in state $s^{'} \in \mathcal{S}$ with probability $\mathcal{T}(s, \mathrm{a}, s')$ and takes an immediate reward $\mathcal{R}(s, \mathrm{a})$. The discount factor $0 \leq \gamma \leq 1$ determines how much are immediate rewards favored over future rewards. If $\gamma = 0$ the agent becomes "myopic", taking only the actions that will maximize its immediate reward. As γ approaches 1, the agents becomes more far-sighted, taking actions that maximize its long term reward.

A solution to an MDP consists of a policy, which is basically a mapping from the state space $\mathcal{S}$ to the action space $\mathcal{A}$, so the agent decides its action by choosing the optimal action for the current state regardless of its past history (Markov property). Ultimately, the solution maximizes the expected total discounted reward:

$$E = \sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, \mathrm{a}_t)$$

Notice how γ makes E finite as it approaches zero given enough timesteps, and thus makes this definition appropriate even for a potentially infinite horizon.

An example of a small MDP can be seen in Figure 2.2. This MDP has two states (high, low), two possible actions (wait, search) when in state high and three (wait, search, recharge) when in state low. There are three different rewards (r_{wait},r_{search}, -3, 0) and the arrows depict the transition function. For example when in state "high" and the robot does the action "search", it lands in state "high" with probability α or in state "low" with probability $(1 - \alpha)$.

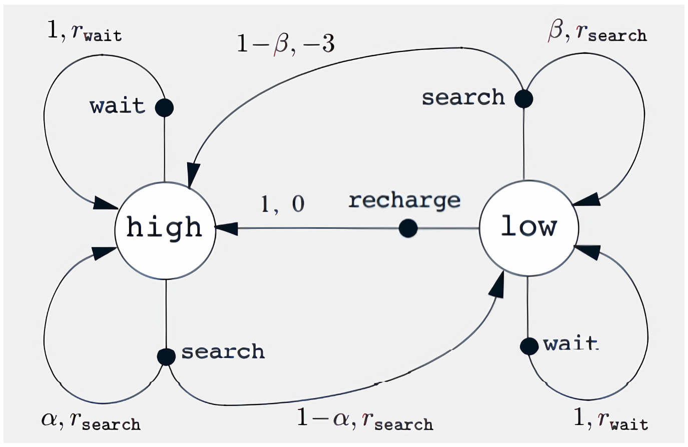


Figure 2.2: Recycling Robot MDP (from [Sutton and Barto, 2018])

2.2.1 Collaborative Multi-Agent MDP

As an MDP is normally used for settings with only one agent, we present an extension for multiagent collaboration settings, namely Collaborative Multi-Agent MDP (CMMDP) [Guestrin, 2003]. CMMDP is structured for systems where there are multiple agents each one with its own set of actions and state variables and each one receives its own observations and rewards, but all agents need to collaborate towards their goal. This can be modeled as a 6-tuple $(\tau, A, \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}, \mathcal{R})$ where:

- τ = 0,1,2... is the timestep
- A is the set of agents, e.g. for 3 agents $A = (\alpha_1, \alpha_2, \alpha_3)$
- $\mathcal{S}_i$ is the set of environmental states for each agent i, the global state $\mathcal{S}$ is the cross product of all $\mathcal{S}_i$, e.g. for 3 agents the global state is $\mathcal{S}_1$x$\mathcal{S}_2$x$\mathcal{S}_3$
- $\mathcal{A}_i$ is the set of actions for each agent i, the joint action on timestep τ for agent i is $\mathbf{a}_i^\tau$
- $\mathcal{T}(s, \text{a}, s')$ is the transition function from a global state $\boldsymbol{s}$ to a global state $\boldsymbol{s'}$ given a joint action $\mathbf{a}$
- $\mathcal{R}_i(s, \text{a})$ is the reward function that provides each agent i with a reward r_i^τ on timestep τ, the global reward can be regarded as the sum of all rewards

2.2.2 Partially Observable MDP

A partially observable MDP (POMDP) is an MDP generalization for partially observable environments. This means that the environmental variables cannot be fully observed. A POMDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma)$ where [Kaelbling et al., 1998]:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{T}(s, \text{a}, s')$ is the transition function

- $\mathcal{R}(s, \mathrm{a})$ is the reward function
- Ω is a finite set of observations
- $\mathcal{O}$ is the observation function
- γ is the discount factor

The $\mathcal{S}, \mathcal{A}, \mathcal{T}(s, \mathrm{a}, s'), \mathcal{R}(s, \mathrm{a})$ elements are the same as before. The difference is that by lying in a state s and taking an action a, the agent makes an observation $o \in \Omega$ with probability $\mathcal{O}(s, \mathrm{a}, o)$. The discount factor γ, $0 \leq \gamma \leq 1$ determines how much are immediate rewards favored over future rewards. If $\gamma = 0$ the agent becomes "myopic", taking only the actions that will maximize its immediate reward. As γ approaches 1, the agent becomes more far-sighted taking actions that maximize its long term reward.
The solution to a POMDP is a policy which maps an optimal action to each state and maximizes the expected total discounted reward, just as in MDP.

2.2.3 Mixed Observability

It is not that rare to have a setting where some elements of the environment are fully observable, while some others are only partially observable. This directs to mixed observability. As a consequence, each state s consists of two parts, one fully observable and one partially observable, such as $s = (\mathcal{X}, \mathcal{Y})$, where $\mathcal{X}$ is the fully observable part and $\mathcal{Y}$ the partially observable one.

Belief-State MDP

By introducing the belief notation to a POMDP, we are able to transform it to an MDP with a continuous state space, by maintaining a belief-state $b(s)$. Formally, a belief-state $b(s)$ is a function that maps each possible state to a probability and it can be proven to be a sufficient statistic which encapsulates the history of the agent [Poupart, 2005]. Like in MDP and POMDP, belief-state MDP can be modeled as a tuple ($\mathcal{B}$, $\mathcal{A}$, τ, ρ, γ) where:

- $\mathcal{B}$ is an infinite set of belief states
- $\mathcal{A}$ is a finite set of actions
- $\tau(b, \mathrm{a}, b')$ is the transition function between belief states
- $\rho(b, \mathrm{a})$ is the reward function
- γ is the discount factor

, where the reward and the transition functions can be computed using the functions of the underlying POMDP:

$$\rho(b, \mathrm{a}) = \sum_{s \in \mathcal{S}} b(s) \mathcal{R}(s, \mathrm{a})$$

$$\tau(b, \mathrm{a}, b^{'}) = \sum_{o \in \Omega} P_r(b^{'}|b, \mathrm{a}, o) P_r(o|\mathrm{a}, b)$$

given that $P_r(o|\mathrm{a}, b) = \sum_{s' \in \mathcal{S}} \mathcal{O}(o|s', \mathrm{a}) \sum_{s \in \mathcal{S}} \mathcal{T}(s'|s, \mathrm{a}) b(s)$ and that $P_r(b'|b, \mathrm{a}, o)$ is 1 if by updating belief state b with action a and observation o we get belief state b', and 0 otherwise.

We can update $b(s)$ everytime we make an action and receive an observation by using the following formula:

$$b_{t+1}^{\mathrm{a},o}(s) = \eta \mathcal{O}(s, \mathrm{a}, o) \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \mathrm{a}, s') b_t(s')$$

, where $\eta = \dfrac{1}{P_r(o|\mathrm{a}, b)}$ is a normalizing constant. This way, we make a more accurate estimation of what the true world state is each time we receive an observation.

Since we maintain a belief state instead of using states, the optimal policy consists of a mapping from a belief state to an optimal action.

The main problem with this format is that in most cases an exact solution is intractable. That is, mainly, due to the "curse of dimensionality", since the size of the belief space B grows exponentially with the size of the state space $|\mathcal{S}|$. We address a way to deal with this issue in the next subsection.

Particle Filtering

Here we present the technique of particle filtering [Silver and Veness, 2010; Thrun, 2000], which can be used to approximate the belief described in the previous subsection. Instead of representing belief as a distribution over all possible states, we can represent it as a set of particles, with each particle being a possible state. In case of mixed observability, each particle corresponds to a possible outcome for the partially observable part of the state only. With that said, if we use K particles, then our belief is:

$$B(s, h_t) = \frac{1}{K} \sum_{i=1}^{K} \delta_{sB_t^i}$$

, where $\delta_{ss'}$ is the Kronecker delta function.

To update our belief when we make an observation of the environment we use a black-box simulator $\mathcal{G}$ of the world. This simulator receives the current state s and an action a as inputs and returns a new state s' and an observation o as if we made the action in the real environment. So when we make a real action and a real observation we pass each particle from B_t through the black box, if the state and the observation returned by $\mathcal{G}$ are the same as the real ones, we sustain the tested particle and insert it in our new set of particles B_{t+1}. We stop this process when we have K particles. This method approaches the true belief if K is sufficiently large.

A complication of this method is that after many belief updates, particle deprivation can occur. To mitigate this, we can simply add artificial noise to each particle to reinvigorate our set.

2.3 Q-Learning

Reinforcement Learning (RL) is an area of Machine Learning (ML) concerned with the maximization of the cumulative reward of an agent, and is quite often used to solve (PO)MDPs with unknown components.

Although RL is a rich field with a wide range of algorithms, we will focus on the Q-learning algorithm.

Q-learning [Watkins and Dayan, 1992] is a model free (does not require a model of the environment) RL algorithm which guides an agent to its next action, based on the agent's current state. To achieve this, it maintains a Q-table ($\mathcal{S}$x$\mathcal{A}$) which holds a Q-value for each state-action pair that represents its quality. The core of the algorithm is a Bellman equation as an update:

$$Q(s, \mathrm{a}) = Q(s, \mathrm{a}) + \alpha(r + \gamma \max_{\mathrm{a}} Q(s^{'}, \mathrm{a}) - Q(s, \mathrm{a})) \tag{2.1}$$

The update takes place in each step, after the agent has made an action a which led to a new state s' and an immediate reward r that depends on the current state and action. The term $\max_{\mathrm{a}} Q(s', \mathrm{a})$ depicts the maximum reward we can get from the future state and makes the agent choose actions that maximize its total reward in long term. The degree of how much future rewards matter is tuned by the parameter $\gamma \in [0, 1]$, as it is also mentioned in the POMDP Section 2.2.2. The $\alpha \in [0, 1]$ parameter is the learning rate of the algorithm, for environments with high stochasticity a small α would be more suitable, while in more deterministic environments a higher one would be better, in the case of a fully deterministic environment the optimal value is 1. The intuition behind α is how much we value new information compared to older.

One central issue for the proper functioning of Q-learning is sufficient exploration, and in particular achieving a balance between exploitation and exploration, that is the balance between exploiting the current best known solution versus exploring new options to hopefully find a better policy than the best known so far. One approach, which we also adopt in this thesis, is the so-called ϵ-greedy exploration, in which the agent at each step chooses either to explore using a random action with a small probability ϵ or to exploit the best action (with the highest Q-value) with probability $1 - \epsilon$. This will help the agent get unstuck from a possible local optimum by keep searching for a better strategy.

Algorithm 1 Q-learning

```
Initialize Q($s$,a)
**for** each episode **do**
  initialize $s$
  **repeat** {for each step in episode}
    choose a$^{'}$ for $s^{'}$ using policy derived from $Q$
    take action a and observe $r$ and $s^{'}$
    $Q(s, \mathrm{a}) \leftarrow Q(s, \mathrm{a}) + \alpha(r + \gamma max_{\mathrm{a}'} Q^{'}(s^{'}, \mathrm{a}^{'}) - Q(s, \mathrm{a}))$
    $s \leftarrow s'$
  **until** $s$ is terminal
**end for**
```

2.3.1 SAS-Q-Learning

In [Boutilier et al., 2018], the authors address the issue of not having all actions always available due to stochasticity. For such environments, they introduce a framework, namely Stochastic Action Set MDP (SAS-MDP), and present a Q-learning variant named SAS-Q-learning, among others, as an algorithm to tackle this kind of problems. It is also clarified that the set of available actions witnessed each time are independent of the agents history.

According to the said work, we can achieve convergence of the Q-values as long as the optimal action chosen at each step is the maximizing action of the available actions, and the Q-value update is done using the next state optimal action considering only the next state available actions as well. Putting this in mathematical terms, we need to select the optimal action as $arg \max_{\mathrm{a} \in \mathrm{A}_t} Q(s, \mathrm{a})$, where A_t is the set of valid actions at timestep t, and the update function:

$$Q(s_t, \mathrm{a}_t) = Q(s_t, \mathrm{a}_t) + a(R + \gamma max_{\mathrm{a}_{t+1} \in \mathrm{A}_{t+1}} Q(s_{t+1}, \mathrm{a}_{t+1}) - Q(s_t, \mathrm{a}_t))$$

is used. Of course, the Q-value of an action is independent to the other available actions. The only limitation for this technique to work, is that the available actions at each step must be provided additionally. Essentially, SAS-Q-learning ranks the actions available based on their respective Q-values and chooses the highest ranking action. It should also be mentioned that, [Chandak et al., 2020] comment that SAS-Q-learning can be unstable in practice.

While our domain does not have SAS, since the set of valid actions at each timestep depends on the agent's history and is computed in a deterministic way, we still employ this technique (with additional details given in Section 3.5 and Section 4.3) and achieve good results as seen in Section 4.4.

2.4 Coordination Graphs

In order to reduce complexity and make our problem more scalable, we make use of Coordination Graphs (CGs) [Guestrin et al., 2002]. This framework, consists of an undirected graph G=(V,E) in which each node $i \in V$ represents an agent, and each edge $(i, j) \in E$ depicts a dependency between two nodes i and j. This assumes that each agent i affects, with its actions, a subset $\Gamma(i) \subseteq V \setminus i$ of the remaining agents, which forms the "neighborhood" of agent i. The agents which are connected must cooperate on the action selection process.

Via CGs, one can convert the problem of global optimization to multiple smaller problems of local optimization. More specifically, the global payoff $u(\mathbf{a})$ of the joint action $\mathbf{a}$ can be decomposed to a linear combination of local payoffs:

$$u(\mathbf{a}) = \sum_{i=1}^{n} f_i(\mathbf{a}_i) \tag{2.2}$$

, where $\mathbf{a}_i$ is the set of i's action and the actions of its neighbors $\Gamma(i)$. If we assume that the payoff functions are described over two agents at most, we can further decompose the utility function as:

$$u(\mathbf{a}) = \sum_{i \in V} f_i(\mathrm{a}_i) + \sum_{(i,j) \in E} f_{ij}(\mathrm{a}_i, \mathrm{a}_j) \tag{2.3}$$

, where f_i is the payoff function regarding an agent's individual action, while f_{ij} is the local payoff function corresponding to a pair of actions of two coordinating agents. Note that the assumption of at most two agents per payoff function does not restrict this framework because each agent can have multiple dependencies.
In Figure 2.3 we can see an example of a CG of 6 agents with their dependencies and payoff functions.

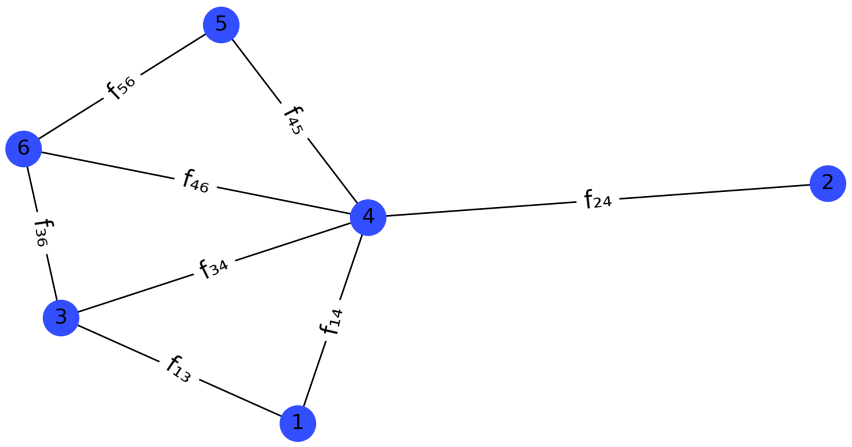


Figure 2.3: Coordination Graph of 6 agents - payoff functions f_{ij} between each pair of agents is depicted

One method to find a maximizing joint action $\mathbf{a}^*$ by utilizing the CG structure, is the Variable Elimination algorithm (VE) [Guestrin et al., 2002]. Although VE is an exact algorithm and obtains the optimal joint action, it does not scale well with the density of the CG. In fact, the complexity grows exponentially, making VE infeasible for a CG with many agents and dependencies [Kok and Vlassis, 2006]. In the next section, we describe a much faster (albeit approximate) algorithm for our problem.

2.5 Decentralized Max-Plus Algorithm

Besides VE, another way to compute the joint action $\mathbf{a}$ that scales linearly with the number of agents and exploits the nature of a CG, is Max-Plus [Kok and Vlassis, 2006]. Max-Plus is a message passing algorithm, which requires each agent to exchange messages with its neighbors over each edge, and propagates the payoff. In particular, each agent i sends a message:

$$\mu_{ij}(\mathrm{a}_j) = max_{\mathrm{a}_i}\{f_i(\mathrm{a}_i) + f_{ij}(\mathrm{a}_i, \mathrm{a}_j) + \sum_{k\in\Gamma(i)\setminus j} \mu_{ki}(\mathrm{a}_i)\} + c_{ij} \tag{2.4}$$

to each of its neighbors $j \in \Gamma(i)$ in the CG. Note that c_{ij} is a normalizing constant and the term $\sum_{k\in\Gamma(i)\setminus j} \mu_{ki}(a_i)$ sums all incoming messages except that from agent j. For agent i each message μ_{ij} maps each action a_j of its neighbor j to a real value that quantifies the effect of j's action a_j to the best payoff that agent i observes for the total utility of all other incoming messages and the corresponding local utility functions. Normally, all agents keep exchanging messages based on the CG they form, until the messages converge or until an external signal is received. Bear in mind that before convergence, μ_{ij} is an approximation of the conditional payoff, since it uses the incoming-unconverged messages from its neighbors.

The nature of the Max-Plus algorithm enables it to be implemented in an anytime form. That means that the agents will keep sending messages and store the best action found so far. So upon the termination, it will report the best joint action found. Although this is an approximate algorithm, and does not guarantee that the optimal action will be found like VE, it is superior in terms of speed at completion and convergence.

Regarding convergence guarantees, max-plus is guaranteed to converge for acyclic graphs, namely trees, after a finite number of iterations. For graphs that contain cycles, max-plus is not theoretically proven to converge, but in practice it has been used in such problems successfully. A key issue is that an agent's outgoing messages can ultimately be incoming messages, due to cycles. This can lead to the values rising up to very large numbers. A way to mitigate this problem, is by subtracting the average of all outgoing messages of agent i to all its neighbors $j \in \Gamma(i)$, that can be achieved by using the normalizing constant like:

$$c_{ij} = -\frac{1}{|A_k|} \sum_k \mu_{ik}(\mathrm{a}_k)$$

with $|A_k|$ being the number of different actions a_k.

Essentially, for an agent i, the incoming message μ_{ji} is an assessment of how good each of its actions are, from the perspective of neighboring agent j. Following this we derive that:

$$g_i(\mathrm{a}_i) = f_i(\mathrm{a}_i) + \sum_{j \in \Gamma(i)} \mu_{ji}(\mathrm{a}_i)$$

Ultimately, $g_i(\mathrm{a}_i)$ is being used by each agent to update its current optimal action a'_i, where $\mathrm{a}'_i = arg\,max_{\mathrm{a}_i} g_i(\mathrm{a}_i)$. As is evident, $g_i(\mathrm{a}_i)$ corresponds to each agent's individual utility function $f_i(\mathrm{a}_i)$ plus the sum of incoming messages for action a_i.

Although max-plus can work as a centralized algorithm, it is much more efficient to implement it in a distributed manner. Unfortunately, this yields two main problems, concerning the global payoff and the evaluation process, since several changing attributes can no longer be considered common knowledge among the agents. To mitigate this, each agent must maintain a representative of the global payoff value as well as be able to tell when to unilaterally start the evaluation process. Subsequently, when certain circumstances are met, and an agent thinks that starting an evaluation is worthwhile, it triggers a chain reaction by sending an *evaluation* message to all of its neighbors on a prefixed spanning tree (ST). When an agent receives an *evaluation* message, it locks its best individual action and propagates the message to its children according to the ST. If the agent has no children, which means it is a leaf node in the ST, it computes its participation in the global payoff and propagates it back to its parent by sending an *accumulate_payoff* message. When an agent receives an *accumulate_payoff* message, it adds the payoff of its child to its own and propagates it up to the root. When the root is reached, the payoff of the joint action has been calculated through propagation and a new sequence to determine if it is better than the previous stored is initiated. For that matter, when an agent receives a *global_payoff* message, it compares the payoff of the current optimal action to that of the previous (which is already stored). If the global payoff is improved, the agent stores the new payoff as the highest attained so far.

Algorithm 2 depicts the pseudocode for Max-Plus.

2.6 SparseQ: Coordinated Multi-agent Reinforcement Learning

Here we present the Sparse Cooperative Q-learning algorithm, or simply SparseQ [Kok and Vlassis, 2006]. SparseQ is a cooperative multi-agent RL framework that utilizes CGs. For this purpose, we first have to approximate the global Q-function by decomposing it to a linear combination of local Q-functions. In the work [Kok and Vlassis, 2006], the authors achieve this with two different decompositions, one agent-based and one edge-based. The former associates a Q-value Q_i to each agent i, while the latter associates a Q-value Q_{ij} to each edge between two agents i, j.

2.6.1 Agent-based decomposition

According to the agent-based decomposition, each agent i is associated with a local Q-function $Q_i(\boldsymbol{s}_i, \mathbf{a}_i)$, as seen in Figure 2.4. As the Q-functions reference a CG, the set of actions $\mathbf{a}_i$ and the subset of the global state $\boldsymbol{s}_i$ relevant to agent i. So instead of using the global joint action and the global state of the environment, we use only the states and actions of agent i and its neighbors. To compute the global Q-function, we simply add all the local Q-functions $Q(\boldsymbol{s}, \mathbf{a}) = \sum_{i=1}^{n} Q_i(\boldsymbol{s}_i, \mathbf{a}_i)$. With this, we can rewrite equation (2.1):

$$\sum_{i=1}^{n} Q_i(\boldsymbol{s}_i, \mathbf{a}_i) = \sum_{i=1}^{n} Q_i(\boldsymbol{s}_i, \mathbf{a}_i) + \alpha[\sum_{i=1}^{n} R_i(\boldsymbol{s}, \mathbf{a}) + \gamma \sum_{i=1}^{n} Q_i(\boldsymbol{s}', \mathbf{a}_i^*) - \sum_{i=1}^{n} Q_i(\boldsymbol{s}_i, \mathbf{a}_i)] \tag{2.5}$$

We can approximate the optimal actions $\mathbf{a}_i^*$ in state $\boldsymbol{s}'$ by using the VE algorithm. Therefore, from (2.5) we get a local update for each local Q-function, as:

$$Q_i(\boldsymbol{s}_i, \mathbf{a}_i) = Q_i(\boldsymbol{s}_i, \mathbf{a}_i) + \alpha[R_i(\boldsymbol{s}, \mathbf{a}) + \gamma Q_i(\boldsymbol{s}_i^{'}, \mathbf{a}_i^*) - Q_i(\boldsymbol{s}_i, \mathbf{a}_i)] \tag{2.6}$$

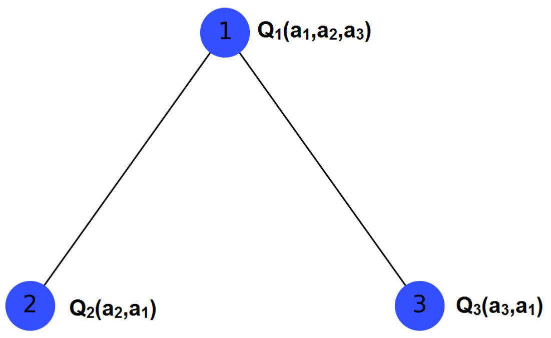


Figure 2.4: Agent-based decomposition on a CG of 3 agents

Clearly, this decomposition scales exponentially with the number of neighbors for each agent. To mitigate this, each edge, instead of agent, can be associated with a pairwise Q-function to achieve linear scaling, leading to an edge-based decomposition.

2.6.2 Edge-based decomposition

According to the edge-based decomposition, each local Q-function Q_{ij} represents a dependency on the CG between two agents i and j and maps their combined state $\boldsymbol{s_{ij}}$ and both of their actions a_i and a_j to a real value $Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j)$. The sum of all local functions approximates the global Q-function:

$$Q(\boldsymbol{s}, \mathbf{a}) = \sum_{(i,j)\in E} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) \tag{2.7}$$

where E is the set of all edges of the CG. An example of an edge-based decomposition on a CG can be seen in Figure 2.5.

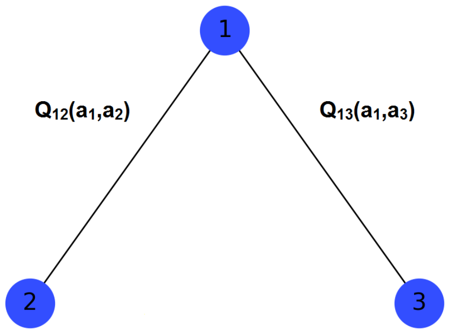


Figure 2.5: Edge-based decomposition on a CG of 3 agents

A problem with this decomposition would be to find the optimal joint action that maximizes the global Q-function by maximizing each local Q_{ij} function. Thankfully, since each Q_{ij} is a pairwise function, we can employ the max-plus algorithm described earlier to do that quickly and efficiently. What we do is simply use the Q_{ij} function as the payoff function, but since in the Q_{ij} function exactly two agents take part, we can slightly alter the messages μ_{ij}:

$$\mu_{ij}(\mathrm{a}_j) = max_{\mathrm{a}_i}\{Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) + \sum_{k\in\Gamma(i)\backslash j} \mu_{ki}(\mathrm{a}_i)\} + c_{ij} \tag{2.8}$$

and the g_i function:

$$g_i(\mathrm{a}_i) = \sum_{j\in\Gamma(i)} \mu_{ji}(\mathrm{a}_i) \tag{2.9}$$

as in [Vlassis, 2007].

There are two methods of updating the Q_{ij} functions after each joint action. The first one is agent-based and the second one is edge-based.

agent-based update

In order to identify the agent-based update for the edge-based decomposition, we define the local function Q_i for each agent i, as:

$$Q_i(\boldsymbol{s}_i, \mathbf{a}_i) = \frac{1}{2} \sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) \tag{2.10}$$

We basically assume that each pair of agents, who are dependent to each other, contribute equally. With that noted, we can naturally obtain from Equation 2.6:

$$\frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) = \frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) + \alpha[R_i(\boldsymbol{s}, \mathbf{a}) + \gamma Q_i(\boldsymbol{s}_i^{'}, \mathbf{a}_i^*) - Q_i(\boldsymbol{s}_i, \mathbf{a}_i)] \tag{2.11}$$

by substitution.

In order to derive a local update for each Q-function, Q_{ij}, we rewrite the temporal-difference error as:

$$R_i(\boldsymbol{s}, \mathbf{a}) + \gamma Q_i(\boldsymbol{s}_i^{'}, \mathbf{a}_i^*) - Q_i(\boldsymbol{s}_i, \mathbf{a}_i) = \sum_{j\in\Gamma(i)} \frac{R_i(\boldsymbol{s}, \mathbf{a}) + \gamma Q_i(\boldsymbol{s}_i^{'}, \mathbf{a}_i^*) - Q_i(\boldsymbol{s}_i, \mathbf{a}_i)}{|\Gamma(i)|} \tag{2.12}$$

and by substituting 2.11 to 2.10 and adding the contribution of each agent to the edge, we get the local update method:

$$Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) = Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) + \alpha \sum_{k\in\{i,j\}} \frac{R_k(\boldsymbol{s}, \mathbf{a}) + \gamma Q_k(\boldsymbol{s}_k^{'}, \mathbf{a}_k^*) - Q_k(\boldsymbol{s}_k, \mathbf{a}_k)}{|\Gamma(k)|} \tag{2.13}$$

During the update, the temporal-difference error is backpropagated from the two agents forming the edge, while utilizing all of the specific agents' dependencies to do so. Max-plus can be used to attain the optimal actions at each state.

edge-based update

In order to obtain the edge-based update rule, we substitute (2.10) to (2.5) and get:

$$\begin{aligned}\tfrac{1}{2}\textstyle\sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) &= \tfrac{1}{2}\textstyle\sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) + \\ &\alpha[\textstyle\sum_{j\in\Gamma(i)} \frac{R_i(\boldsymbol{s},\mathbf{a})}{|\Gamma(i)|} + \gamma\tfrac{1}{2}\textstyle\sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}^{'}, \mathrm{a}_i^*, \mathrm{a}_j^*) - \tfrac{1}{2}\textstyle\sum_{j\in\Gamma(i)} Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j)]\end{aligned} \tag{2.14}$$

But since we need a method to update each local Q-function Q_{ij} individually we remove the sums from (2.14) and because agent i and agent j both update the function Q_{ij}, we can add the two parts to finally obtain the update rule for each Q_{ij}:

$$Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) = Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) + \alpha[\frac{R_i(\boldsymbol{s}, \mathbf{a})}{|\Gamma(i)|} + \frac{R_j(\boldsymbol{s}, \mathbf{a})}{|\Gamma(j)|} + \gamma Q_{ij}(\boldsymbol{s}_{ij}^{'}, \mathrm{a}_i^*, \mathrm{a}_j^*) - Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j)] \tag{2.15}$$

Note that each agent's reward is normalized according to the number of its neighbors, as we assume that they all contribute equally. In addition, the optimal joint action $(\mathrm{a}_i^*, \mathrm{a}_j^*)$ in the next state $\boldsymbol{s}_{ij}^{'}$ can be computed using max-plus.

2.7 Partially Observable Monte-Carlo Planning

In this section we examine the Partially Observable Monte-Carlo Planning (POMCP) [Silver and Veness, 2010] algorithm in order to portray the OP as a tree associating each node with a history of actions and observations, and represent the not observable congestion in nodes as a set of particles (belief). The set of particles is initialized by selecting random hidden states uniformly, over all possible hidden states. Then, simulations are performed to generate a policy for the current step, given a particle sampled from the current history's belief. For a history outside the tree, a uniformly random policy can be used as an estimated evaluation. After doing an action in the real world and acquiring a real observation, the tree is pruned suitably and the belief is updated by moving particles to the new root. In case of a lack of particles, new fabricated ones are made by applying local transformations on current particles. Intuitively, the authors combine two known methods; MCTS and Monte-Carlo updates for the belief state. The two techniques are merged in such an efficient way, that makes the algorithm thrive even in large POMDPs [Silver and Veness, 2010]. It is also important to note that POMCP does not need to know the dynamics of the environment rather than use a black-box simulator $\mathcal{G}$, which is utilized to generate series of actions, observations and rewards.

2.7.1 Partially Observable UCT

In POMCP, we use Upper Confidence bounds applied to Trees (UCT) [Kocsis and Szepesvári, 2006], with a few tweaks to deal with the partially observable environment. UCT maintains a tree to represent the environment and analyzes the most promising actions. Each node of the tree $T(h)$ represents a history h and contains information $\langle N(h), V(h)\rangle$, where $N(h)$ is the counter for the number of times history h has been visited and $V(h)$ is the value of the history. Each node is initialized to $\langle 0, 0\rangle$ but domain-specific knowledge can be exploited during initialization in order to direct the search to more promising subtrees.

At the start of each simulation we use a sampled state from the root's belief $\mathcal{B}$ as the initial state. Just like the fully observable UCT, actions are selected using a heuristic and specifically UCB1 [Auer et al., 2002]:

$$V(h\mathrm{a}) = V(h\mathrm{a}) + c * \sqrt{\frac{log(N(h))}{N(h\mathrm{a})}}$$

$N(h\mathrm{a})$ is the number of times the node with history ha has been visited, and N(h) is the number of times the node's parent has been visited. The whole term $c * \sqrt{\frac{log(N(h))}{N(h\mathrm{a})}}$ gives an exploration bonus that is higher for rarely visited nodes. This bonus is tuned by parameter c, the higher c is, the more explorative the algorithm. When $c = 0$ the algorithm acts greedily, exploiting the best action every time. The parameter c can be given a value based on the specific problem, for example a higher c could be used for a large environment as exploration would be crucial.

While selecting actions/nodes using the above heuristic, when a leaf is reached we use a rollout policy based on the given history, $\pi(h, \mathrm{a})$ to choose actions. Exactly one node is added to the tree after a simulation.

When a number of simulations has been ran, the action with the highest value V(ha) is selected.

2.7.2 POMCP

POMCP [Castellini et al., 2019; Silver and Veness, 2010] is essentially the PO-UCT algorithm with the addition of Monte-Carlo belief state updates , and that is achieved by using the same simulations for both. We add an extra element to each node of our tree, namely a set of belief particles $\mathcal{B}(h)$. So now each node of the tree is represented as $T(h) = \langle N(h), V(h), \mathcal{B}(h) \rangle$.

POMCP consists of three main functions: *Search*, *Simulate* and *Rollout*. In *Search*, given the current history h_t, we sample a starting state as described for PO-UCT or, if history is empty, we sample uniformly from all possible states/particles. During simulation, if the corresponding node exists in the tree, we use UCB1 to choose an action a and then pass it along with the current state through our black box simulator to obtain a new state, an observation and an immediate reward. Then, *Simulate* is called recursively until we reach a maximum depth or a leaf of the tree. The relative nodes' counters and values are updated properly and the total reward (immediate reward + delayed reward) is returned. On the contrary, if the node doesn't exist in the tree, we expand it by all available actions and return the result of *Rollout*.

In *Rollout* we do a random walk starting from a leaf of the tree recursively, in similar fashion with *Simulate*, and return the total reward. Fundamentally, this means taking an initial guess for the value of a node.

The above repeats for a predefined number of times or until a time limit is reached. When the search procedure completes, we choose the action from the root with the highest value and receive an observation from the environment. After that, the root moves from $T(h_t)$ to $T(h_t a_t o_t)$ and we sample a new state from $\mathcal{B}(h_t a_t o_t)$ to run the algorithm again. As expected, the rest of the tree is pruned since it can no longer be reached.

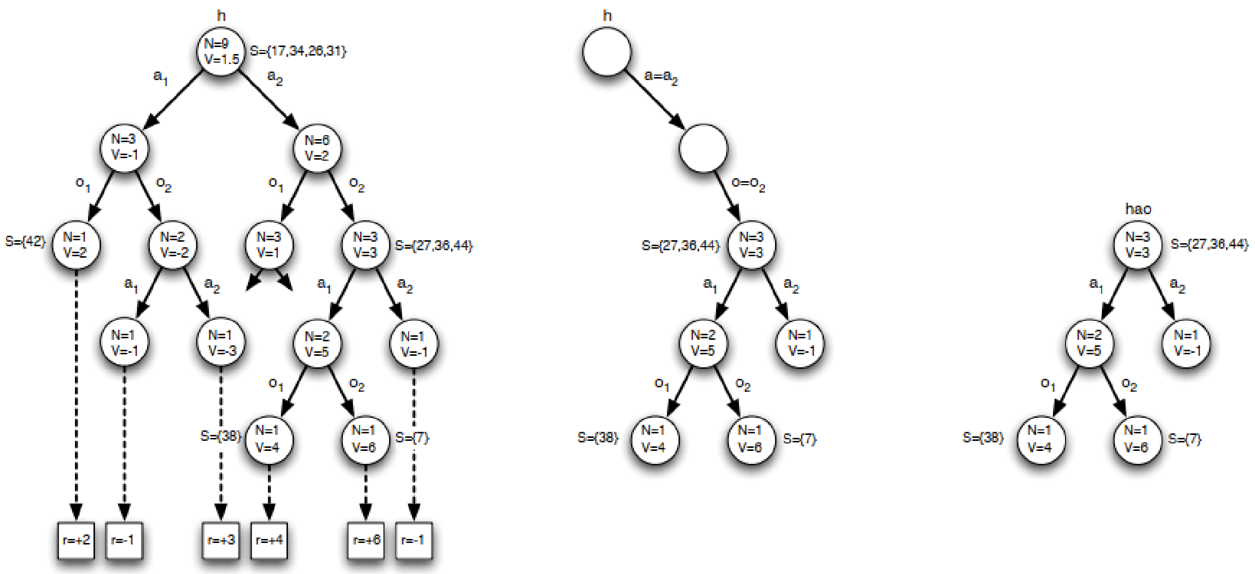


Figure 2.6: An example of POMCP, in an environment with 2 actions (from [Silver and Veness, 2010])

Figure 2.6 depicts a tree maintained by POMCP (left), as well as making a real action followed by a real observation (middle) which leads to the moving of the root and the pruning of the tree (right). The pseudocode of POMCP can be seen at Algorithm 3.

After pruning the tree and setting the new root, there is a chance that its number of particles is very low. A difficulty like this can be quite common in large POMDPs that can also have a large number of available actions, as in our problem. To alleviate this issue, a *particle reinvigoration* technique can be used that adds noisy particles in the root's belief. Such particles are obtained by applying a local transformation to an already existing particle.

In [Castellini et al., 2019] they extend POMCP to exploit prior knowledge in problems that pose constraints. They do so with the use of two methods; Constraint Network Initialization (CNI) and Markov Random Field Initialization (MRFI), in order to alter the process of belief particle initialization as well as the process of particle reinvigoration. CNI uses a hard constraint representation whereas MRFI uses a probabilistic one. Even though we do not explicitly use the techniques described in their work, the constraints, of our problem in study, are taken into consideration during the initialization and reinvigoration processes, as described in detail in subsection 3.3.1 and subsection 3.3.2 respectively.

2.8 Related Work

Our work in this thesis was mainly motivated by three papers:

1. DIRECT: A scalable approach for route guidance in selfish orienteering problems [Varakantham et al., 2015]

2. A reinforcement learning framework for trajectory prediction under uncertainty and budget constraint [Le et al., 2016]

3. Multi-agent Orienteering Problem with Time-Dependent Capacity Constraints [Chen et al., 2014]

In [Varakantham et al., 2015], the authors combine the OP and Selfish Routing to create a Selfish OP setting. With that said, there are multiple agents traversing a graph while respecting individual limitations and act selfishly. There also are different types of agents with each type having its own latency function which show the tolerance of each type's towards congestion on an edge. On top of it, each agent type has a set of specific nodes that aspires to visit, and a budget that consists of the restrictions. Budget could be the minimum number of nodes from each type's desired nodes or the maximum time an agent has in its disposal, among others. Eventually, each agent's goal is to formulate a path that has the minimum total latency from start to end node, and satisfy the budget limitation. For a solution they compute and employ Nash equilibrium. In order to do so, and since a direct approach cannot scale, they utilize a scalable non-pairwise formulation to enforce the equilibrium condition, and introduce a master-slave decomposition (DIRECT) approach to compute an approximate equilibrium. And, actually, despite associating congestion with edges, for one of the experimentation instances, they associate congestion with nodes , just like in this work. They also state that instead of the goal being the finding of path(s) for each agent that minimize(s) the latency based on the agent's type, a goal of maximizing utility could also be used. It is clear that SeOP proves to be relatively close to this work both intuitively and technically. A number of differences can be summed up to the key difference of all the agents being identical and equivalent in this thesis. This, in our perception, makes our setting a bit more competitive, as each agent has the exact same constraints and is interested at each node as much as the others. In addition, there are no latency functions, as agents are penalized even for the smallest fraction of congestion (i.e. two agents visiting the same node). Also, the solution techniques we adopt are entirely different.

In (2) above, the authors use RL to model the agent's sequential decision making while incorporating stochasticity of rewards and a budget constraint. The goal is to predict the following trajectory, given the current trajectory and observed ones of other similar agents. Their algorithm was used in a real theme park to guide tourists, with each trajectory being a sequence of location visits. Their approach is basically a combination of learning and prediction. With respect to learning, they divide the agents into clusters and use Hidden Markov Models to transform states into trajectory of sequences, then model them as an MDP and use Inverse RL to model the utility function of each state. Regarding prediction, they use Viterbi's algorithm to compute the most plausible sequence of states for the observed trajectory and the agent's type, then they predict the next sequence of visits that can meet the goal. Comparing [Le et al., 2016] with this thesis, a few differences stand out. First, they utilize different types of agents, while in this thesis all agents are identical. Secondly, they use inverse RL to learn the reward function, while in this thesis the reward of visiting a node is the node's fixed score with a congestion-driven discount. Finally, they use a partial trajectory of an agent to predict the destinations to be visited and the reward to be expected in the future while in our setting, each agent has to compose a path from the starting node to the ending node with no prior information required.

Finally, in [Chen et al., 2014] the authors introduce a multi-agent modification to the OP with capacitated nodes. Each node has a capacity and exceeding it will lead to waiting times for each agent involved. The agents affect each other when they arrive in the same node simultaneously and the paper uses game theory, and more specifically fictitious play [Brown, 1951; Lambert Iii et al., 2005], to find good joint actions. In contrast to that approach, we do not use capacitated nodes, instead each node can host any number of agents but at the same time penalizes them depending on the congestion of each node at each timestep. Also, instead of using time penalties to punish the agents, we use score discounts. Again, the solution techniques we use in this thesis are entirely different to those of [Chen et al., 2014].

Other related work can be seen in [Best et al., 2020; Gama and Fernandes, 2020] as well, where they employ Monte-Carlo tree search and RL, respectively, to tackle OP variants.

Algorithm 2 MAXPLUS [Kok and Vlassis, 2006]

```
$threshold = 0, a_i\_lock =$ **false**,$g = 0, p = 0, m = -\infty$
**Function** $maxplus$(msg,ST,CG)
    **if** $msg.type == maxplus$ **then**
        $value = \max(msg.value)$
        $j \sim \Gamma(i)$ {prioritize neighbors to whom we have not sent yet}
        **for all** $a_j \in j.actions$ {use $j$'s available actions, as in SAS-Q-learning} **do**
            $\mu_{ij}(a_j) = \max_{a_i}(Q_{ij}(a_i, a_j) + \sum_{k \in \Gamma(i) \setminus j} \mu_{ki}(a_i)) + c_{ij}$
        **end for**
        $send(\mu_{ij}, j)$
        **if** $value > threshold$ **and** $receivedAll(\Gamma(i), maxplus)$ **then**
            $send(evaluate(), i)$ {to self}
            $threshold = 1.02 * value$
        **end if**
    **else if** $msg.type == evaluate$ **then**
        **if** $a_i\_lock ==$**false then**
            $a_i^{'} = arg\max_{a_i}(\sum_{j \in \Gamma(i)} \mu_{ji}(a_i))$
            $a_i\_lock =$ **true**
        **else**
            **return**
        **end if**
        $send(evaluate(), i)$ {to all neighbors except j}
        **if** $ST.isLeaf(i)$ **then**
            $send(accumulate(0), i)$
        **end if**
    **else if** $msg.type == accumulate$ **then**
        $p_i = p_i + msg.value$
        **if** $receivedAll$(ST.children($i$), $accumulate$) **then**
            $g_i = \frac{\sum_{j \in \Gamma(i)} Q_{ij}(a_i^{'}, a_j^{'})}{2}$
            **if** $i == ST.root$ **then**
                $send(global\_payoff(g_i + p_i)$,$i)$
            **else**
                $send$(accumulate($g_i + p_i$),ST.parent($i$))
            **end if**
        **end if**
    **else if** $msg.type == global\_payoff$ **then**
        **if** $msg.value > m$ **then**
            $a_i^* = a_i^{'}$
            $m = g$
        **end if**
        **for all** child $\in$ ST.children($i$) **do**
            $send(global\_payoff(g)$,child)
        **end for**
        $a_i\_lock =$**false**
    **end if**
```

Algorithm 3 POMCP [Silver and Veness (2010)]

```
Function search(h, n):
    $i = 0$
    while $i < n$ do
        if $h == []$ then
                $s \sim \mathcal{I}$ {sample from uniform distribution}
        else
                $s \sim \mathcal{B}(h)$ {sample from current belief}
        end if
        $simulate(s, h, 0)$
        $i = i + 1$
    end while
    return $arg\max_{\mathrm{a}} V(h, \mathrm{a})$
Function simulate(s, h, depth):
    $valid\_actions = compute\_actions(h)$
    if $valid\_actions == []$ then
        return 0
    end if
    if $\gamma^{depth} < \epsilon$ then
        return 0
    end if
    if $isLeaf(h)$ then
        for all $\mathrm{a} \in valid\_actions$ do
                $T(h, \mathrm{a}) = (0, 0, [])$
        end for
        return $rollout(s, h, depth)$
    end if
    $\mathrm{a} = arg\max_{\mathrm{a}} V(h\mathrm{a}) + c\sqrt{\frac{logN(h)}{N(ha}}$
    $s', o, r \sim \mathcal{G}(s, \mathrm{a})$
    $R = r + \gamma simulate(s', hao, depth + 1)$
    $B(h).append(s)$
    $N(h) = N(h) + 1$
    $N(h, \mathrm{a}) = N(h, \mathrm{a}) + 1$
    $V(h, \mathrm{a}) = V(h, \mathrm{a}) + \frac{R - V(h,\mathrm{a})}{N(h,\mathrm{a})}$
    return R
Function rollout(s, h, depth):
    $valid\_actions = compute\_actions(h)$
    if $valid\_actions == []$ then
        return 0
    end if
    if $\gamma^{depth} < \epsilon$ then
        return 0
    end if
    if informed() == true then
        $favored\_actions = best\_actions(h, valid\_actions)$
    else
        $favored\_actions = valid\_actions$
    end if
    $\mathrm{a} \sim \mathcal{I}(favored\_actions)$ {Sample uniformly from $favored\_actions$}
    $s', o, r \sim \mathcal{G}(s, \mathrm{a})$
    return $r + \gamma rollout(s', hao, depth + 1)$
```

Chapter 3

Our Approach

In this chapter we provide a formulation of our setting and explain the core of our work, namely the use of SparseQ and POMCP to address the multiagent Orienteering Problem. To be able to do so, we also had to come up with two distinct formulations of our setting, first as a Collaborative Multiagent MDP and then as a POMDP.

3.1 Our Extension

In our modification of OP we adopt a multi-agent setting which can be thought as TOP but with more tours, as well as the addition of congestion in each node. Ultimately, this extension can be used in any OP variant. More rigorously, we have a set of P agents $p = 1, \ldots, P$ on top of an OP setting and its attributes. At timestep 0, all agents coexist in the starting node and are called to choose which node they will visit in the next timestep. Once the timestep advances each agent is moved to its desired node and receives a discounted score dependent on the number of agents visiting the same node on the same timestep. Eventually, all agents after a finite number of steps and while abiding by the constraints and limitations of the underlying OP instance, land in the ending node. Naturally, when an agent visits the ending node, it can no longer move and has completed its run. Essentially, the goal of this variation is to maximize the summation of the total scores, similarly to the OP variants that require the computation of multiple paths. Mathematically the objective function goal is:

$$max \sum_{p=1}^{P} \sum_{i=2}^{N-1} y_{ip\tau} S_i d_f^{(Q_{i\tau}-1)}$$

, where $y_{ip\tau}$ is 1 if agent p is in node i during timestep τ and 0 otherwise, S_i is the score of node i, $Q_{i\tau}$ equals the number of agents in node i during timestep τ, N is the number of nodes and lastly, $d_f \in (0, 1)$ is the congestion coefficient [1]. A d_f close to 0 is very punishing, while a d_f close to 1 is very forgiving.

The main difference regarding the mathematical formulation of this setting compared to that of other variants is the goal function already stated. The rest of rules can be easily derived from the already existing ones. As a matter of reference we provide the formulation for this extension on the TOP format (see Section 2.1.1)

[1]discount severity due to congestion

given the extra decision variable x_{ijp} on top of $y_{ip\tau}$, S_i, $Q_{i\tau}$ and d_f, where x_{ijp} is 1 when a visit to node i is followed by a visit in node j in agent p's path. In addition, variable u_{ip} node's i place in agent's p path (i.e. the order in which node i is visited), and t_{ij} is the time needed to travel to node j starting from node i.

1. $\sum_{p=1}^{P} \sum_{j=2}^{N} x_{1jp} = \sum_{p=1}^{P} \sum_{i=1}^{N-1} x_{iNp} = P$
2. $\sum_{p=1}^{P} y_{kp\tau} \leq 1, \forall k = 2, \ldots, N-1, \forall \tau = 1, 2, \ldots$
3. $\sum_{i=1}^{N-1} x_{ikp} = \sum_{j=2}^{N} x_{kjp}, \forall k = 2, \ldots, N-1, \forall p = 1, \ldots, P$
4. $\sum_{i=1}^{N-1} \sum_{j=2}^{N} t_{ij} x_{ijp} \leq T_{max}, \forall p = 1, \ldots, P$
5. $2 \leq u_{ip} \leq N, \forall i = 2, \ldots, N, \forall p = 1, \ldots, P$
6. $u_{ip} - u_{j\alpha} + 1 \leq (N-1)(1 - x_{ijp}), \forall j = 2, \ldots, N, \forall p = 1, \ldots, P$

Constraint (1) ensures that each agent's path has to start and end in the starting and goal node respectively by ensuring that there is only one node that succeeds node 1 and only one node that precedes node N, constraint (2) ensures that each node must be visited at most once by ensuring that no node can show up more than once in a path, equation (3) makes certain that the path is connected with respect to the order of the nodes by ensuring that the number of nodes preceding a node k must be equal to the number of nodes succeeding node k, constraint (4) makes sure that each agent's path is limited by the time budget T_{max} by comparing it with the total travel time needed to traverse the path, and finally, constraints (5) and (6) prevent subtours. Constraint (6) specifically, ensures that if node j succeeds node i in agent's p path ($x_{ijp} = 1$), then the order of visit to node j must be greater than that of node i, $u_{jp} \geq u_{ip} + 1$.

The notion of computing multiple paths that is included in some OP variants (Section 2.1), is replaced here by the existence of multiple agents.

The above constraints hold for the OP and TOP files, for the MCTOPMTW files there need to be added rules for the extra limitations, such as the knapsack constraints and the extra duration associated with each node.

An example application of this setting could be the Tourist Trip Design Problem (TTDP) [Gavalas et al., 2014; Gavalas et al., 2015], where each agent in our setting represents a group of tourists and the nodes represent the Points of Interest (POI). The score of each node may represent the quality or significance of each POI. Our setting could be used in order to avoid overcrowding, which can occur when multiple groups of tourists (i.e. agents) visit the same POI at the same timestep.

3.2 Applying SparseQ

Regarding SparseQ, we use the edge-based decomposition and both the edge-based and agent-based update methods as they are already described in Section 2.6.2. The agent-based decomposition of the Q-values was deemed inappropriate for our problem since it scales exponentially with the number of neighbors and can be proven cumbersome.

To apply SparseQ, we model our setting as a CMMDP [Guestrin, 2003]:

- $\tau = 0, 1, 2, \ldots$, the timestep
- $P = \{p_1, p_2, \ldots, p_n\}$, a group of n agents
- $\mathcal{S} = \{\mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n\}$, the global state as a cross-product of all agents' states, the state an agent is in, is the set of the current valid actions
- $\mathcal{A}_{p_i}$, is the set of actions for each agent p_i, the size of the action set varies since some actions may be unavailable due to the constraints applied
- $\mathcal{T}$, the state transitions function, since the transitions are completely deterministic, the transition from a state to another will be either 1 or 0 depending on the limitations
- $\mathcal{R}$, the reward function, the reward for an action a is the node's discounted score. The function used is $d_f^{|Q_{n\tau}|-1} score(\mathrm{a})$, where $Q_{n\tau}$ is the set of agents in node n at timestep τ and $0 < d_f < 1$ is the congestion coefficient that represents the severity of the congestion at the node. Evidently, when $|Q_{n\tau}| = 1$ the reward is the original score of node n

Regarding the reward function, $\mathcal{R}$, we empirically chose to use $d_f = 0.8$.

We construct a CG for any given number of agents. It would be ideal for the CG to be a complete graph, so that every agent would have to collaborate with every other agent. Unfortunately, this would be really slow in practice and max-plus would have a hard time converging. For example, for 6 agents the CG would have a branching factor of 5 and 15 edges in total, in addition to the large size of the state and action spaces early on. To mitigate this, we add an edge with a probability, while ensuring that the branching factor will not exceed 3. The number 3 was chosen according to the benchmarks presented in [Kok and Vlassis, 2006] for max-plus for different branching factors.

We are in the special case of not having all the actions available at any time, due to the nature of the OP. For this matter, we use the set of valid actions at a timestep to represent the state. Certainly, this will lead in the blow up of the state space. To try to alleviate this issue along with the large set of actions early on, we initialize every Q_{ij}-value according to the undiscounted score of the resulting nodes:

$$Q_{ij}(\boldsymbol{s}_{ij}, \mathrm{a}_i, \mathrm{a}_j) = score(\mathrm{a}_i) + score(\mathrm{a}_j)$$

This will encourage each agent to favor most rewarding actions first.

3.2.1 Applying Max-Plus

We apply max-plus as in Algorithm 2, with a few remarks. First, we assume that each agent is able to know its neighbors available actions. Secondly, an agent reaching the goal node will still keep coordinating with its neighbors in order to suggest their actions' values, until its neighbors have reached the ending node as well.

3.3 Applying POMCP

We apply the vanilla POMCP algorithm as introduced in Section 2.7.2 (as in the original paper [Silver and Veness, 2010]), and also use POMCP with the addition of domain-specific knowledge as a separate algorithm. In order to so, we model our problem as a POMDP:

- $\mathcal{S}$: Each state is the unfactored representation of the congestion level at each node along with the history of actions. Congestion is partially observable as each agent can only observe the congestion in the current node while the agent's history is fully observable, e.g. for a problem with 100 nodes we have a 100-tuple of integers and a tuple of the actions done so far as the state.

- $\mathcal{A}$: each node is an action. Since constraints limit our actions, the set of viable ones at each time may vary, e.g. for a problem instance of 50 nodes set of actions is 50 at start but keeps decaying as the agent does more actions.

- $\mathcal{O}$: an observation is the number of agents in a node, for example, for 4 agents the set of possible observations is (0,1,2,3).

- $\mathcal{Z}$: we model the observation function as a Cauchy distribution

- $\mathcal{R}, \mathcal{T}$ are the same as described in Section 3.2.

Each agent runs POMCP independently and tries to maximize its own cumulative reward. A serious problem with this, is that the full history tree will be really large, even for small instances. For a datafile with 50 nodes, the root of the tree will have up to 49 children, and each one of them will have up to 48 children and so on. Essentially, the tree will be extremely dense near the root and sparse near the leaves. This raises the need to utilize a set of preferred actions.

To decide which nodes should be favored we employ a heuristic presented in [Golden et al., 1987]:

$$WR(i) = aSR(i) + bCR(i) + cER(i) \tag{3.1}$$

for each node i, where SR is the ranking of the node depending on its undiscounted score, CR is the ranking of the node depending on its distance from the center of gravity, and ER is the ranking of the node depending on the sum of distances to the start and goal nodes. As the center of gravity we use the average coordinations of all nodes if the agent has no history, otherwise the average coordinations of the nodes visited so far. The resulting ranking WR of all the nodes is the weighted sum of all 3 rankings SR, CR and ER, where $a + b + c = 1$. The weights can be chosen empirically.

We now provide further details on each ranking metric. For further detail about the ranking of each metric; for the metric SR the rank of 1 is assigned to the node that has the largest score and the ranks increase as the scores decrease, for CR rank 1 is assigned to the node that is closer to the center of gravity and the ranks increase as the distances increase, and for ER a rank of 1 is given to the node with the smallest sum of distances from the starting and ending nodes with the ranks increasing as the sum of distances increase. We use the resulting rankings to obtain $WR(i)$, for each node i, and rank all the nodes based on WR in ascending order.

The node with the lowest $WR(i)$ will be the most promising option according to the heuristic. In any case of tie, in any of the aforementioned metrics, we assign the same rank.

To allow for a better understanding of this method, we consider a toy example of 6 nodes. In Table 3.1 we can see each node's information along with the respective ranking for each metric (SR, CR, ER) and finally the final ranking (WR) of the considered nodes. Node 0 and node 5 are the starting and goal node respectively, and thus were not considered. Each node's information consists of its identification code (id), its coordinations (x and y) and its score. The values of $CR(id)$, $ER(id)$ and $WR(id)$ are also provided to make the table more comprehensive. In particular; $CR(id)$ is the distance of node id from the center of gravity , $ER(id)$ is the sum of distances from the starting and goal nodes and $WR(id)$ is the result of Equation (3.1) for each node. For reference, the coordinations of the center of gravity is $(2, 3.5)$ and the parameters used regarding the Equation (3.1) were $a = 0.3$, $b = 0.35$ and $c = 0.35$. As is evident, node 1 is proposed as the best option since it achieves rank 1 in WR, despite having the lowest score.

Table 3.1: Example use of the heuristic

id	x	y	$score$	SR	$CR(id)$	CR	$ER(id)$	ER	$WR(id)$	WR
0	0	0	0	-	-	-	-	-	-	-
1	2	3	10	4	0.5	1	7.2	1	1.9	**1**
2	1	3	20	1	1.11	2	7.63	3	2.05	**3**
3	2	4	12	3	0.5	1	7.63	3	2.3	**4**
4	3	4	15	2	1.11	2	7.23	2	2	**2**
5	5	5	0	-	-	-	-	-	-	-

The obvious way would be to initialize preferred actions with an optimistic value V_{init} and a non-zero initial counter N_{init}, as well as sampling from this set of actions during rollouts, as in [Silver and Veness, 2010]. But, since we have the notion of congestion, and the agents have no communication whatsoever, we need a more subtle approach. Ultimately, we choose to do only one part and sample from a set of the 20% of the highest ranking actions, depending on the path so far. We decided to do this with the hope of reducing the percentage of congestion. Since, otherwise, the heuristic would dictate very specific actions to each agent and would certainly lead to the agents visiting the same nodes, and thus get discounted rewards. As stated in Section 4.1, we use a version of POMCP with this feature (POMCP-inf) and a version that does not use domain specific information.

Furthermore, to initialize the exploration parameter c, used by the UCB1 heuristic, we run POMCP once and set c according to $c = R_{hi} - R_{lo}$, where R_{hi} is the highest reward returned during the sample run, and R_{lo} the lowest reward returned during the rollouts. We initialize each node equally to $\langle N_{init} = 0, V_{init} = 0\rangle$. In Algorithm 3 a pseudocode for POMCP is provided.

Finally, to model the observations and state transition in the black-box simulator $\mathcal{G}$, we sample from a mixture of $Cauchy$(loc,scale) distributions with $scale = 1$ and:

$$loc = max(\mathcal{O}) * \frac{score(\mathrm{a}_i) - arg\,min_{\mathrm{a}}(score(\mathrm{a}))}{arg\,max_{\mathrm{a}}(score(\mathrm{a})) - arg\,min_{\mathrm{a}}(score(\mathrm{a}))}$$

We decided to use a heavy tailed distribution in order to model the congestion at each node since this type of distributions have successfully been employed to model network traffic [Li, 2018; Paxson and Floyd, 1994]. After testing with different heavy tail distributions (*Weibull*, *Pareto* and *t-distribution* among others) we chose to use the *Cauchy*. As we can see the distribution we sample from depends on the normalized score of the action. Moreover, since it is possible that we sample numbers outside of the set $[min(\mathcal{O}), max(\mathcal{O})]$, we could simply reject any sample that is out of bounds. A more efficient strategy though is that of inverse transform sampling.

In short, we generate a random number $u \sim \mathcal{I}[min(\mathcal{O}), max(\mathcal{O})]$, and we use the inverse of the Cumulative Density Function (CDF) of the *Cauchy* distribution, also known as the quantile function, $F_x^{-1}(X) = loc + scale * tan(\pi * (x - 1/2))$ to calculate $X = F_x^{-1}(u)$. The random variable X is our sample and will inevitably lie in $(min(\mathcal{O}), max(\mathcal{O})]$; we finally round X to the closest integer and thus obtain our sample observation in $[min(\mathcal{O}, max(\mathcal{O})]$.

3.3.1 Particle Initialization

When running POMCP for the first time, running simulations from the root associated with an empty history, we need to initialize a set of particles to represent the initial belief. For this purpose, we have to take into account the constraints of the specific problem in study. With that said, we must sample a particle where the sum of congestion over all nodes equals the total number of agents. Given a set of nodes N and a number of agents A, we choose randomly from a uniform distribution a set of A nodes, N' (each node can be sampled more than once). Subsequently, we set each node's congestion to be equal to the number of times it appears in N'. For example, if we have 6 nodes and 3 agents, a suitable particle could be $\langle 0, 0, 2, 0, 1, 0\rangle$.

3.3.2 Particle Reinvigoration

As mentioned in Section 2.7.2, after doing an action in the real world and obtaining an observation, we move the root and prune the rest of the tree. We also add a number of noisy particles in the new root by applying local transformations to existing ones. To transform a particle, we simply transfer an agent from its respective node to a new one. For the transformation to be considered valid, the new, artificial, particle must be in line with the observation just made. Although with this method, the limitation expressed in Section 3.3.1 (i.e. the sum of congestion over all nodes equals the total number of agents), is adhered, we need to put up two more rules in order to be consistent with the conditions of our problem. To this end, we also ensure that the agents that exist in the goal node in the particle cannot be moved in another node and that if a number n of agents exist in a node i, then the number of agents in that node in the new particle can be at most $N - n - 1$, with N being the total number of agents. The last constraint is natural as agents cannot visit nodes they have already been to in the past. Of course we can choose to do a random (small) number of transformations at each time, instead of just one depending on the number of agents. An example transformation given a particle $\langle 0, 0, 2, 0, 1, 0\rangle$ and an observation 0 at node 2, could be $\langle 0, 0, 1, 1, 1, 0\rangle$. After this step, $k/16$ are added to the new root's belief, where k is the number of simulations ran.

3.4 Applying Q-learning

We use a straightforward application of Q-learning with an ϵ-greedy strategy to our problem as a baseline method. Each agent runs the algorithm independently, having no communication or coordination with the other agents whatsoever. Nevertheless, everyone will still be affected by each others' moves and experience discounts on their individual rewards. Inevitably, this will lead to the agents receiving different rewards for the same action on the same state on different occasions, during training.

3.5 Further Improvements and Relaxations

This problem as presented, naturally has large action and state spaces, even for a single agent, let alone for multiple ones. For example, a graph that consists of only 20 vertices can have up to 20^4 different joint actions for just 4 agents, at each timestep. To mitigate this, we utilize Golden's heuristic as presented in Sections 3.3 and 4.1 in order to consider a subset of most promising actions instead of all available actions. More specifically, we take into account the top 20% if it is more than two times the number of total agents, otherwise the top $(2 * \#agents)$ actions are used. Considering the aforementioned example, the number of distinct joint actions would be up to 8^4, which is a substantial difference compared to 20^4. It should be noted that this method reiterates each time an agent performs an action, as it calls this method based on its own path so far. With this relaxation, the agents can find the optimal paths even though we expect the learning process to be more sensitive. This sensitivity is due to a severely pruned number of actions for each agent, and thus potentially leading to more congested nodes. However, in practice, we obtain a boost in performance.

To deal with the state space size, we employ SAS-Q-learning (Section 2.3.1). Considering the adaption of this technique for SparseQ-edge and SparseQ-agent, each agent simply uses the subset of available actions when running the maxplus algorithm, in order to find the estimated optimal joint action. Again, we assume that the agents know, or at least can communicate, their neighbors' available actions. With these said, we can avoid embedding the set of valid actions in the state, and use just the current node to represent the state instead. Although in [Boutilier et al., 2018] it is stated that the set of valid actions is independent of the agent's history, in our setting it is clear that the valid actions at each step are dependent on the past actions in a deterministic way. Nevertheless, we argue that this choice of ours will work, since each agent can witness different sets of available actions every time it transits to the same state. In the end, this choice is indeed justified by the results. So, for a dataset with 20 nodes, each agent has 20 different states it can be in.

However, when opting to use the current node only in order to represent the state, a problem emerges with respect to the employment of the SparseQ method, as SparseQ cannot readily use this new representation. Specifically, the fact that agents can reach the destination vertex in different timesteps, and each edge's Q-value is updated using the Q-values of the neighbors of the agents forming the edge, leads to inconsistent behavior during training. Of course this problem existed before applying this change, but it did not matter as much when the state space was so large that the negative values affected only a small portion of the total states. With this new change in effect , we apply a simple solution by disconnecting an agent from the CG when it reaches the destination node. When the episode is over the agents form again the original CG. Of course, this is done in a deterministic manner, so as to make sure that the resulting graph will always be the same. Another measure taken is ensuring that there must be at least two nodes in the graph at any time in order to be able to run maxplus.

The above methods are expected to enhance SparseQ-edge and SparseQ-agent, to achieve better results, and on top of that, enable maxplus to run much faster than before, mainly because of the relaxation in the action space. With these modifications, we can use denser CGs, especially for the instances that 4 and 5 agents were used, which in turn will yield additional benefits in the performance of SparseQ algorithms.

Chapter 4

Experimental Evaluation

In this chapter, we present problem instances we experimented with in detail, along with choices and assumptions made, before displaying the test results and concluding with the discussion of them.

4.1 Setup

To test the algorithms we use six data files of three different variants of the OP. The first two are from the TOP variant [Chao, 1993; Chao et al., 1996], the next two is from the TDOPTW [Verbeeck et al., 2017b], and the last two are MCTOPMTW [Souffriau, 2010; Souffriau et al., 2013] datasets.

Regarding the TOP instance (Appendix A and Appendix B), in the first line we are given a single number that represents the number of nodes, in second line the number of paths to be computed, in third line the time budget (tmax). Each remaining line represents the information of a node, giving the x-coordinate, the y-coordinate and the score.

Regarding the TDOP instances (Appendix C and Appendix D), in the first line we are given the number of nodes. Each remaining line represents the information of a node; the node's id, its coordinates, its score, and its opening and closing windows of service. In this one the distances are given on separate files in a time-dependent and time-independent manner respectively. Thus, the problem is essentially interpreted as OPTW. In our case, the time-independent files were used and thus the problem is essentially interpreted as OPTW. It should be highlighted that in these datafiles the distances are not symmetric, that means for two nodes n_i and n_j, $dist(n_i, n_j) = dist(n_j, n_i)$ does not hold necessarily.

Finally, regarding the more complex MCTOPMTW files (Appendix E and Appendix F), in the first line we are given general information; namely the number of paths to be computed, the number of vertices, the (fee) budget limitation and a number of knapsack constraints defining the maximum number of vertices of each type that can be visited. The second line contains the information of the starting point (which is also the ending point) and specifically giving the point's id, its coordinates, the visiting duration, its score, and the opening and closing times of its time window. The starting node's closing time is also the time budget of the instance. Each remaining line represents a regular point and contains the following data: i, x, y, d, s, O_1, O_2, O_3, O_4, C_4, E, b, e_z. Where i is the point's id, x is the x-coordinate, y is the y-coordinate, d is the visiting duration, s is the score, O_1 is the opening time of the first window, O_2 is the opening time of the second window and at the same time the closing time of the first one, O_3 is the opening time of the third window and the closing time of the second one, O_4 is the opening time of the fourth window and the closing time of the third one, C_4 is the closing time of the fourth window, E is a binary value; if 1 the windows used are 1 and 3, otherwise windows 2 and 4 are used, and finally each remaining binary values e_z dictates whether or not the node is of type z. We used only one time window per vertex, with the opening time being the opening of the first time window, and the closing time being the closing of the last time window.

To be more specific, we used p5.3.z and p7.4.k datafiles from [Chao et al., 1996], 50.4.3 and 100.4.3 from [Verbeeck et al., 2017a] and pr01 and c101 from [Souffriau et al., 2013]. p5.3.z has 66 nodes and 5 agents were used, p7.4.k has 102 nodes and 8 agents were used, 50.4.3 has 50 nodes and 4 agents were used, 100.4.3 has 100 nodes and 8 agents were used, pr01 has 48 nodes and 4 agents were used, and c101 has 100 nodes and 8 agents were used. For the file names to be more representative we use the format "type-#nodes-#agents" to rename the files. From now on we refer to p5.3.z as TOP-66-5, p7.4.k as TOP-102-8, 50.4.3 as TDOP-50-4, 100.4.3 as TDOP-100-8, pr01 as MCTOPMTW-48-4 and c101 as MCTOPMTW-100-8.

On every datafile we run SparseQ using the edge-based decomposition of the Q-values and both the edge-based decomposition and agent-based decomposition for the updates, resulting in two algorithms SparseQ-Edge and SparseQ-Agent. We also run classic POMCP and domain informed POMCP-inf where preferred actions are sampled during Rollout, other than sampling from all possible actions. For reference, we also ran the vanilla Q-learning algorithm with each agent independently learning on its own, and a Random algorithm where each agent does a uniformly sampled action from all available actions at each step. In total we tested 6 algorithms, SparseQ-Edge, SparseQ-Agent, POMCP, POMCP-inf, Q-learning and Random.

SparseQ-Edge,SparseQ-Agent and Q-learning, agents were trained for 2000 episodes and tested every 50 episodes, by interrupting the training process to monitor the learning process, with ϵ decaying exponentially from 1.0 to 0.01 and α decaying from 1.0 to 0.1. The discount factor, γ was fixed at 0.9. An episode consists of all the agents doing a complete run and finishing at the goal node and resetting for the next one. The performance depicted in the results tables in Section 4.2 references the ones achieved after the last episode. Specifically for SparseQ-Edge and SparseQ-Agent, we estimate the optimal joint action at each step, by running Max-Plus at most 50 times. Each agent calls for evaluation after it has received at least one Max-Plus message from each neighbor, and afterwards it evaluates only when its maximum score is improved by at least 2%. At this point, we present the fixed CGs used for Max-Plus during testing in Figures 4.1, 4.2 and 4.3. Note that the vertices were connected randomly while ensuring that the branching factor will always lie in [2,3] and using a seed for reproducibility. For reference, the CG of 4 agents has branching factor of 2, the CG of 5 agents a branching factor of 2.4, and the one with 8 agents, has a branching factor of 2.75.

The actual number of agents used here was ultimately chosen with two things in mind (on top of the application on TTDP as described in Section 3.1). First, for the algorithms, and especially Max-Plus, to be able to run in a reasonable time on a regular machine. Second, and to a lesser extend, for the number of agents to be proportionate to the number of total nodes of each dataset (approximately 0.08

agents per node were used). Obviously, this setting can be used, as is or slightly modified, to model other problems as well.

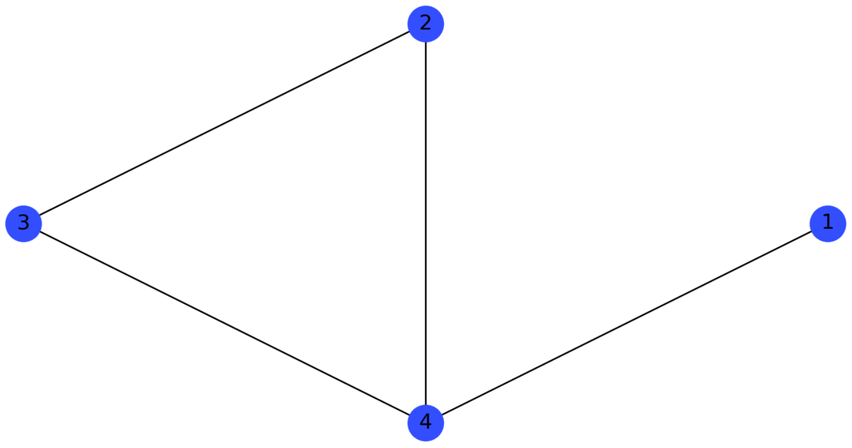


Figure 4.1: Coordination Graph of 4 agents

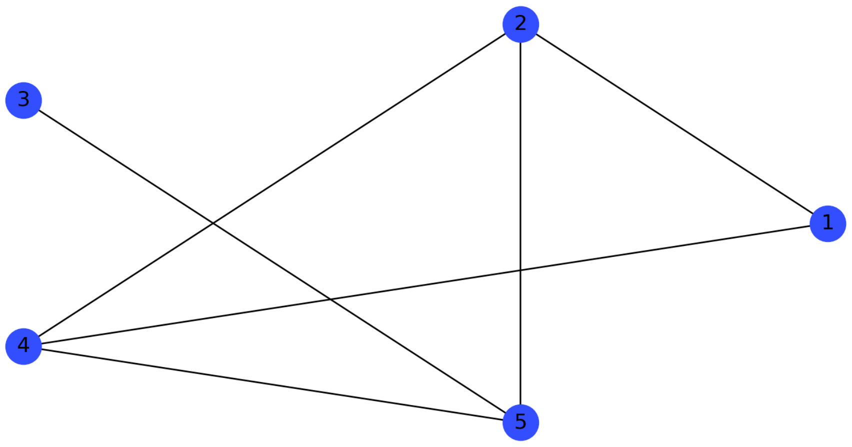


Figure 4.2: Coordination Graph of 5 agents

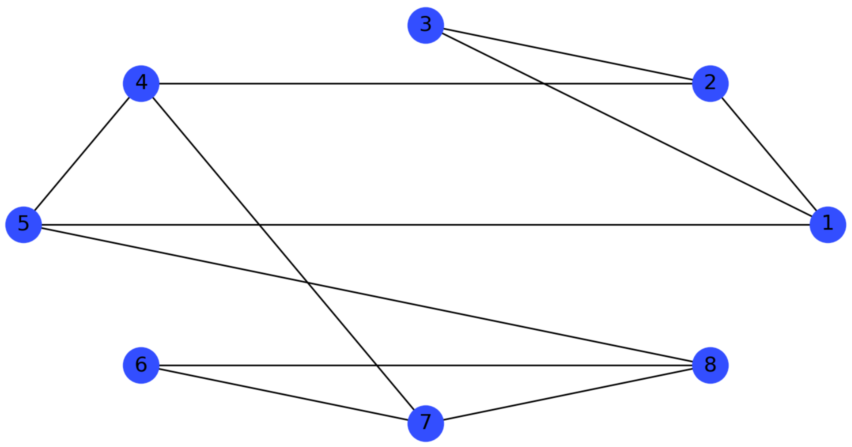


Figure 4.3: Coordination Graph of 8 agents

Regarding POMCP, we run 4000 simulations for each agent independently and before running for the first time, we do 50 sample simulations and 50 sample rollouts to specify the exploration parameter as described in detail in Section 3.3. We fix the parameters γ and ϵ to 0.95 and 0.01 respectively. For the informed version of POMCP, POMCP-inf, we use Golden's heuristic as introduced in Section 3.3, but since this was made for OP and TOP, we introduce some new metrics for the MCTOPMTW files. So, for MCTOPMTW we use:

$$WR(i) = a * SR(i) + b * CR(i) + c * ER(i) + d * DR(i) + e * FR(i) \tag{4.1}$$

for each node i with DR being the duration ranking, and FR being the ranking relative to the fees paid. In both DR and FR, we assign rank 1 to the node with the lowest value and incremental ranks as the values increase. Of course, every other metric (WR, SR, CR, ER) remain the same, as explained in Section 3.3. Had we not modified equation 3.1, agents could end up favoring nodes really costly and ending their runs prematurely. To estimate the best constants a, b, c, d, and e for each problem we should ideally run for different sets and keep the best performing one. As a rule of thumb, we use the normalized counter of distinct ranks for each ranking.

$$a = \frac{|SR|}{|SR| + |CR| + |ER| + |DR| + |FR|}$$

$$b = \frac{|CR|}{|SR| + |CR| + |ER| + |DR| + |FR|}$$

$$c = \frac{|ER|}{|SR| + |CR| + |ER| + |DR| + |FR|}$$

$$d = \frac{|DR|}{|SR| + |CR| + |ER| + |DR| + |FR|}$$

$$e = \frac{|FR|}{|SR| + |CR| + |ER| + |DR| + |FR|}$$

With this technique, we aim to get a good estimation of the ideal (in term of performance) parameters. Intuitively, we ought to give a greater weight to the metrics that offer the better divisibility between the nodes. For example, we would want to give a greater weight to a node ranking higher in a metric that distinguishes all nodes in 20 distinct ranks, over a metric that distinguishes the nodes in just 8 ranks. We do similarly for the TOP files. For the TDOP files the only information provided for each vertex is the score, and there are no coordinations associated with each one. There was an attempt to use the score along with the distance of each node from the last node visited but it produced very poor results during testing and was decided to not use POMCP-inf on these files.

Finally, in Section 4.2 we can see tables with the performance of each algorithm for each problem instance depicting the maximum, minimum and average reward achieved as well as their discounted counterparts, and the average amount of steps the agents did. Furthermore, a box plot graph is provided that serves as a visual representation of the span of discounted rewards, while also depicting the average and median along with possible outliers. This can provide a visual perspective and more information than simply the average, like how the rewards spread between the agents and whether there are heavy overperformers or underperformers. There are also individual figures (specifically Figure 4.4, Figure 4.5, Figure 4.7, Figure 4.8, Figure 4.10 and Figure 4.11) showing the learning process of SparseQ-edge, SparseQ-agent and Q-learning. A good performance of an algorithm would need a high average cumulative reward along with some "uniformity" among the distinct agents' scores; in particular no agent should be performing much worse than the average.

To emphasize; besides the average number of steps for each algorithm, the contents of the tables (Table 4.1, Table 4.2, Table 4.3, Table 4.4, Table 4.5 and Table 4.6), are separated into two sections, one with congestion discounts applied and one without. In the latter we can see the undiscounted maximum total score, the undiscounted minimum total score, and the undiscounted average total score. As the term "undiscounted" suggests, these are the scores the agents would have achieved had it not been for congestion (which occurs when two or more agents visit the same node at the same timestep). Next to them, we can see their discounted counterparts after applying congestion penalties; since when an agent visits a congested node, it receives a percentage of the node's original score, proportionate to the degree of congestion. Naturally, the agents can have (often significant) portions of their scores reduced by congestion. The comparison between undiscounted and discounted scores can serve as a visualization of how well each algorithm performs in terms of congestion control. To be exact, the undiscounted total score of an agent is computed as $\sum_{i \in P} score(i)$ given the agent's path P. The respective undiscounted total score is computed as $\sum_{i \in P, c \in C} 0.8^{c-1} score(i)$ (see the reward function in Section 3.2), given the vector of congestion observed at each section of the path; C, in addition to the path P. Obviously these values can also be calculated in an online manner as the agent traverses the graph.

4.2 Results

Table 4.1: TOP-66-5 results

TOP-66-5 - 5 agents							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	308	220	253.2	315	220	256	10.2
SPARSEQ-Agent	305	208	242.2	305	215	245	9.2
Q-learning	305	238	276.2	305	245	279	10.8
POMCP-inf	557	499.2	**530.12**	575	515	**544**	19.4
POMCP	580	445	519	580	445	519	18
Random	167	99	139.8	175	100	143	7.6

Regarding the first TOP file, TOP-66-5, both POMCP versions outperform by far all other algorithms, scoring more than double the average score of SparseQ-Edge and SparseQ-Agent, as seen in Table 4.1. More specifically, POMCP-inf, does slightly better than classic POMCP, and SparseQ-Edge does better than SparseQ-Agent. Evidently Q-learning performs noticeably better than both SparseQ variants. From the box plot in Figure 4.6 we can see that POMCP-inf and Q-learning present no outliers, and all others present at least one outlier agent. Namely, POMCP and SparseQ-Edge have two outliers each, while SparseQ-Agent has only an overperforming one. We notice that (Figure 4.4) Q-learning's learning curve is smooth enough, while the other two have more bumpy curves but definitely improve over time.

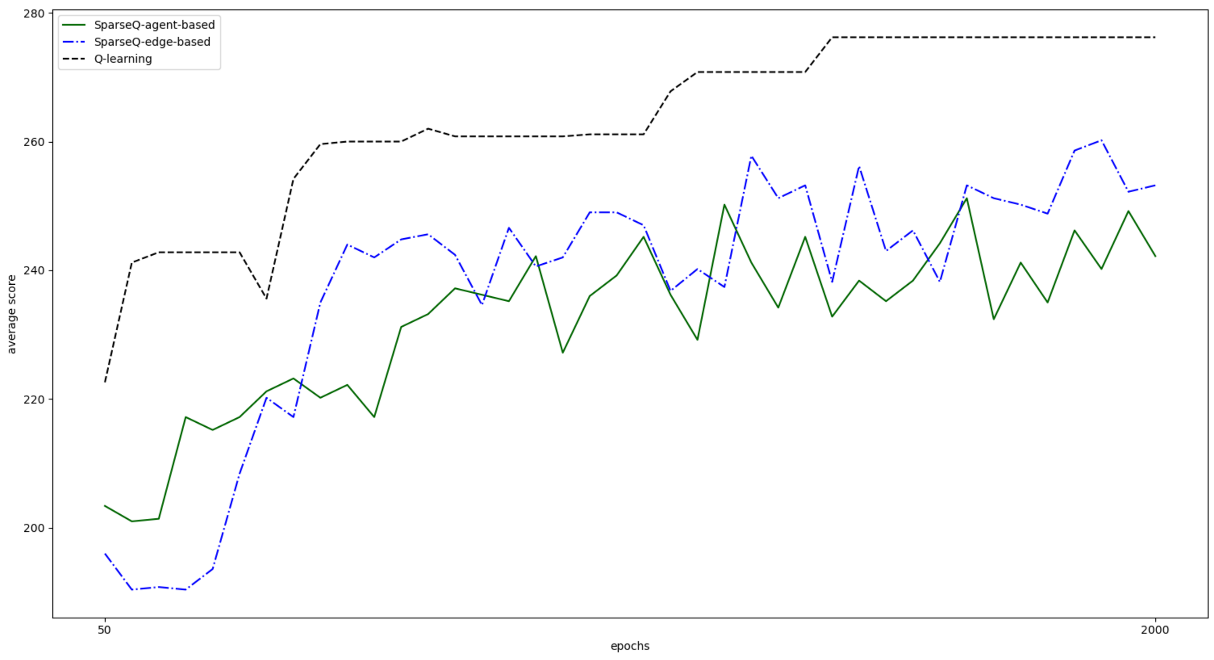


Figure 4.4: Learning curves for TOP-66-5 file

Table 4.2: TOP-102-8 results

TOP-102-8							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	169.24	105.04	135.2	193	136	161.375	7.375
SPARSEQ-Agent	161	109.36	126.23	161	124	150.62	6.875
Q-learning	157.88	140.48	**151.2**	184	151	170.62	8.125
POMCP-inf	178.8	93.83	110.62	193	173	180.625	8.625
POMCP	164.6	125.05	136.82	193	173	**185**	8.875
Random	96	16	47.75	96	16	47.75	3.625

For the second TOP file which has more (102) nodes, TOP-102-8, we get from the results in Table 4.2 that Q-learning achieves the best average total score. A thing worth mentioning is that although POMCP does slightly better than SparseQ-Edge, the latter does almost as well with less congestion, and in fewer average steps. Interestingly enough, both POMCP and POMCP-inf achieve a better undiscounted average score than Q-learning, but lose a significant portion of this due to congestion. This is the reason behind the change in ranking of algorithms between discounted and undiscounted average scores. If we disregard congestion, POMCP performs the best, but since POMCP agents create congestion by visiting the same nodes at the same timestep, they lose a large portion of the total score they would otherwise get. Additionally, each algorithm has at least one outlier, except for SparseQ-Agent as is evident from the relative box plot 4.6. In detail, Q-learning's, POMCP's and POMCP-inf's maximum achieved total scores are outliers whilst SparseQ-Edge has one of each outliers, one for the minimum total score, and one for the maximum one. The learning curves in Figure 4.5 are kind of similar to the ones of file TOP-66-5.

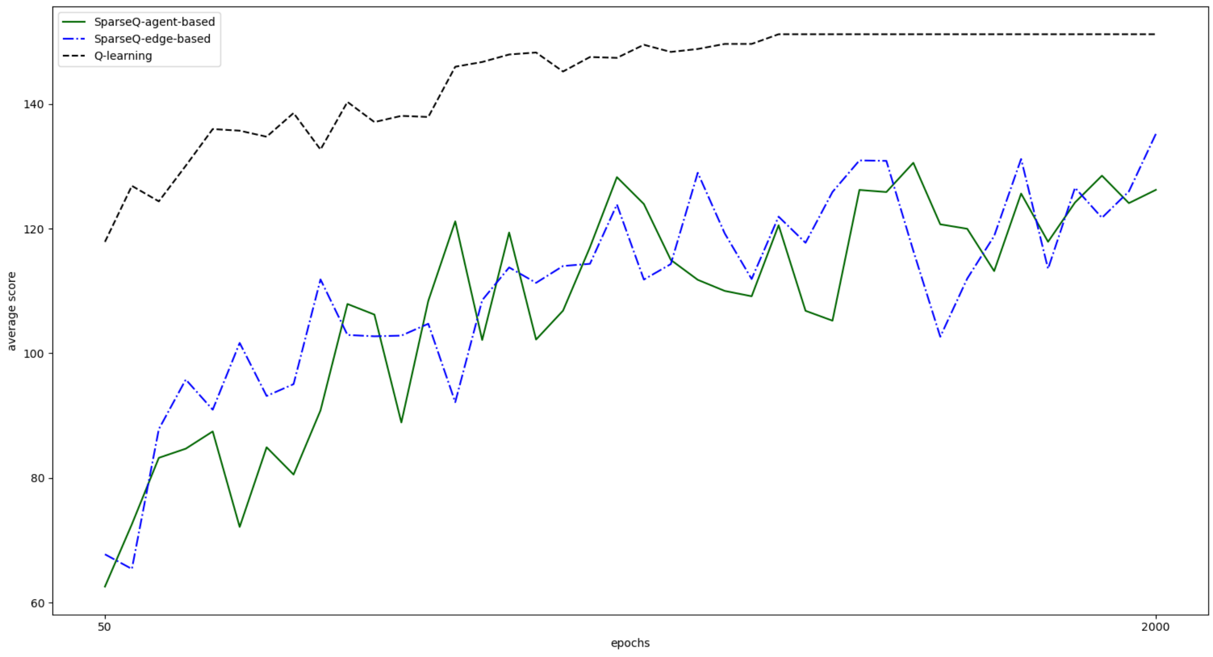


Figure 4.5: Learning curves for TOP-102-8 file

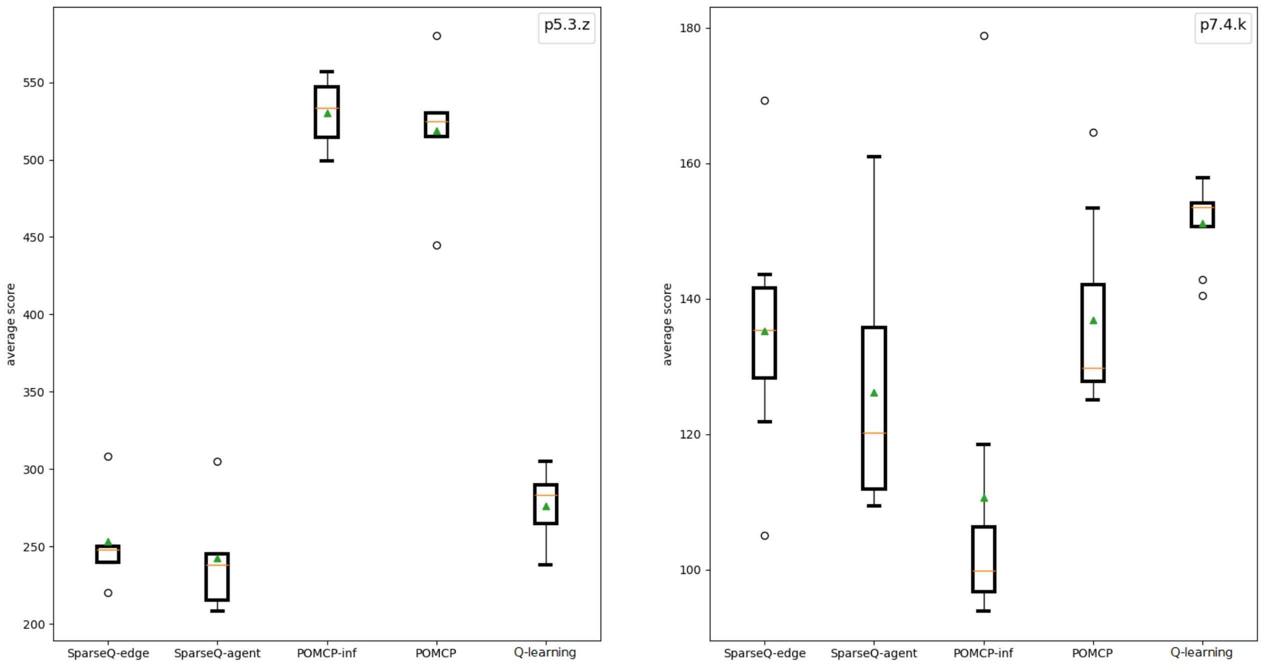


Figure 4.6: Box plots of discounted rewards for TOP-66-5 and TOP-102-8

Concerning the TDOP datafile, TDOP-50-4, and its corresponding Table 4.3, Q-learning achieves the best average score here as well, although POMCP has a

Table 4.3: TDOP-50-4 results

TDOP-50-4							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	173.8	151	164.9	181	151	168.5	6.25
SPARSEQ-Agent	171	116.4	142.2	171	124	146	5.25
Q-learning	207	177	**196.75**	207	177	196.75	7.25
POMCP	197.2	180.32	188.94	232	205	**217.75**	8
Random	119.8	31	81.4	124	31	84	4.75

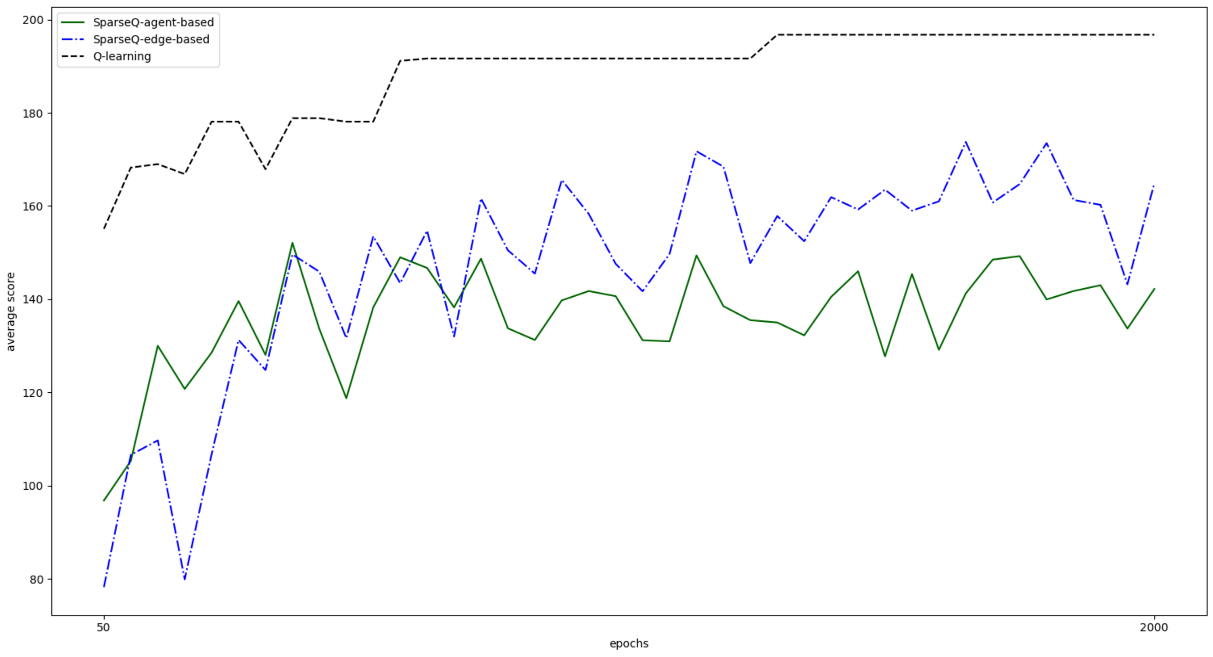


Figure 4.7: Learning curves for TDOP-50-4 file

better undiscounted score, again. Likewise, SparseQ algorithms do a better job at maintaining low levels of congestion, but fall behind in score. The box plot of this file 4.9 shows no outliers in any algorithm.

Table 4.4: TDOP-100-8 results

TDOP-100-8							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	198.2	125	165.07	215	125	181	6.5
SPARSEQ-Agent	218.6	127.48	167.4	234	167	193	6.625
Q-learning	257.2	192.6	220.83	277	208	242.62	8.5
POMCP	286.2	194.15	**232.61**	299	220	**254.75**	9.625
Random	145	28.4	78.69	145	29	79.75	4.875

Regarding dataset TDOP-100-8, POMCP achieves the best performance surpassing that of Q-learning. In addition, POMCP also appears to do better regarding congestion control, as depicted in Table 4.4. SparseQ-Agent and SparseQ-Edge again

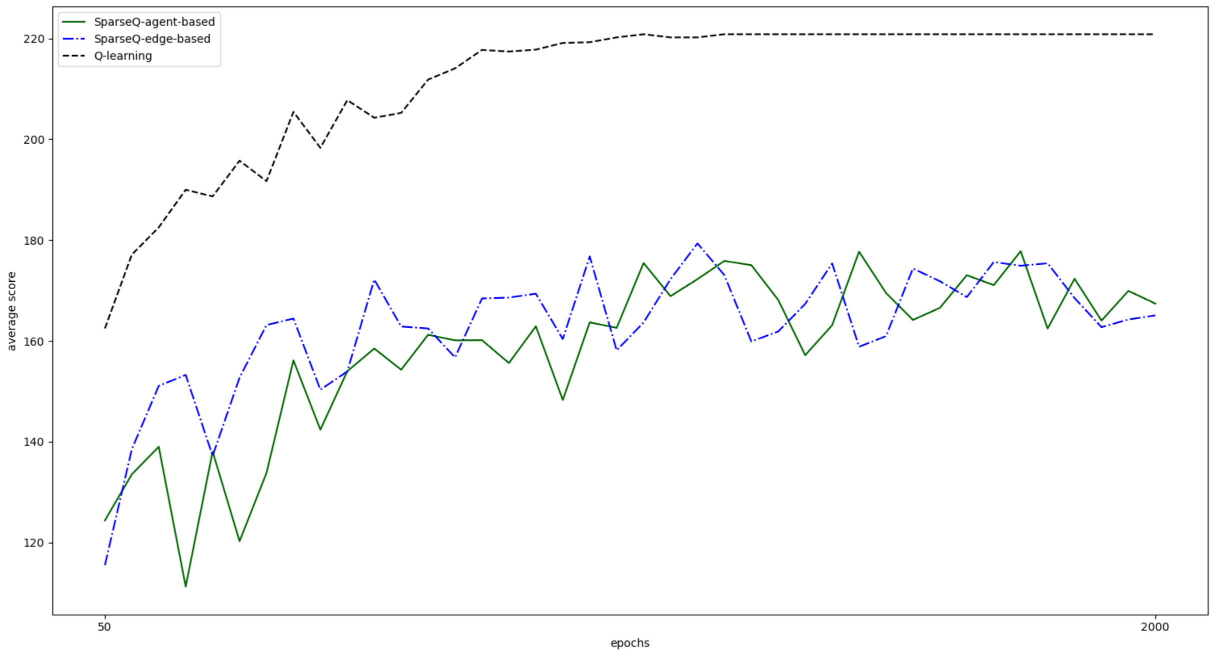


Figure 4.8: Learning curves for TDOP-100-8 file

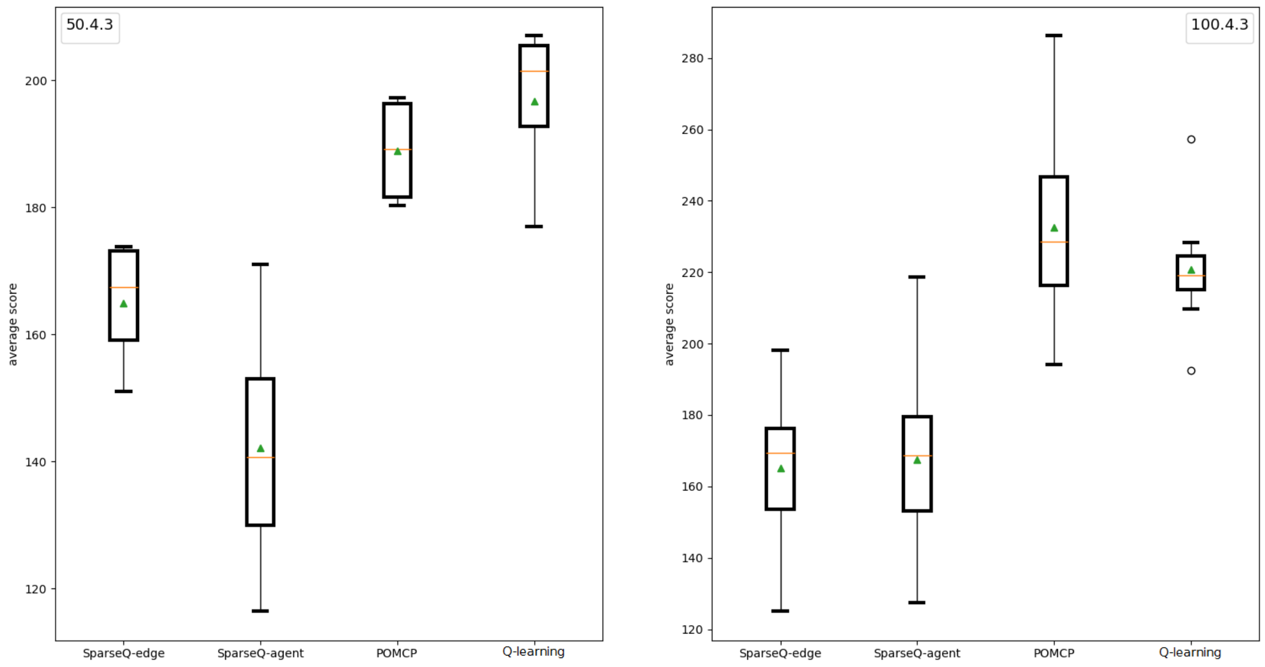


Figure 4.9: Box plots of discounted rewards for TDOP-50-4 and TDOP-100-8

very close to each other in terms of performance, with SparseQ-Agent doing better this time, but nonetheless they both are underperforming in this file too. In the corresponding box plot 4.9 we witness two outliers for Q-learning, and unfortunately an underperforming one too.

Table 4.5: MCTOPMTW-48-4 results

MCTOPMTW-48-4							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	182	106	135.55	187	106	140.25	8
SPARSEQ-Agent	153.6	94	127.5	158	94	131	7.5
Q-learning	220	152	181.5	220	152	181.5	10.75
POMCP-inf	227	184.8	**214.3**	255	217	**239**	13.5
POMCP	189.79	168.6	177.2	240	181	225.25	12.25
Random	90	56.2	71.85	90	59	73.25	6.5

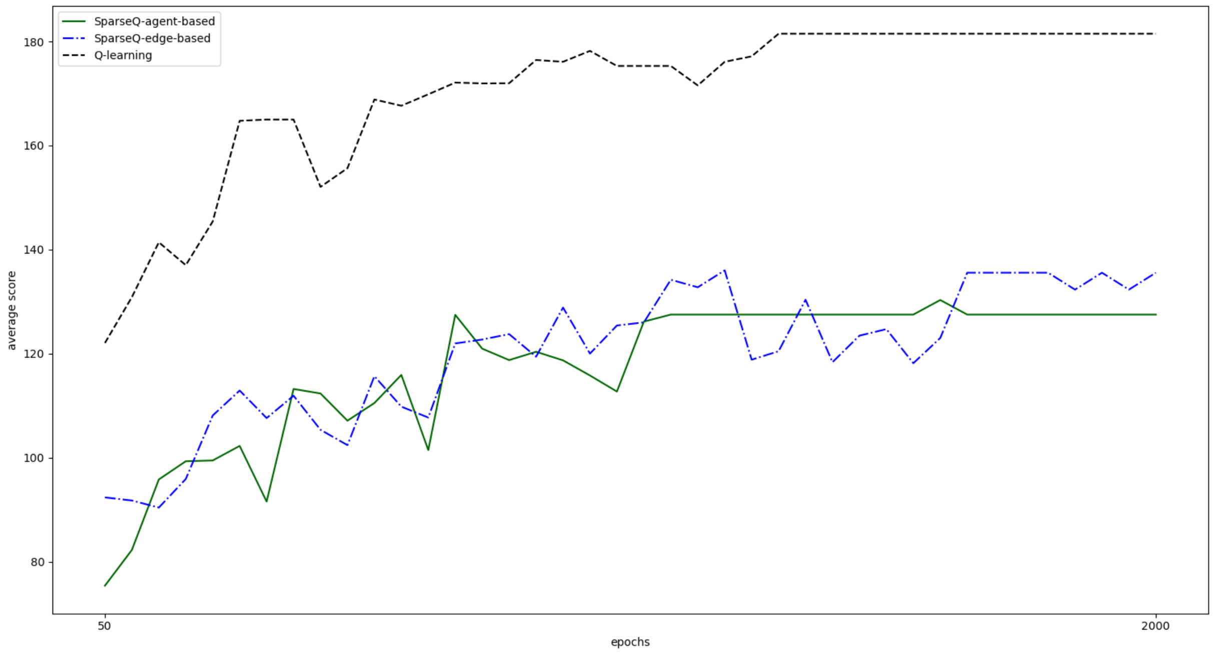


Figure 4.10: Learning curves for MCTOPMTW-48-4 file

With respect to the MCTOPMTW problem instance, MCTOPMTW-48-4, POMCP-inf is the best performing algorithm as we can see from Table 4.5. Specifically, it does significantly better than vanilla POMCP and Q-learning. It is remarkable that even the worst performing POMCP-inf agent is doing better than the average POMCP and Q-learning ones. Both SparseQ agents are once again relatively close with SparseQ-Edge prevailing. A feature that Q-learning prevails in is the congestion levels, as the algorithm attains zero congestion, which can be derived from the fact that the average undiscounted total reward is equal to the discounted one (Table 4.6). SparseQ-edge and SparseQ-agent also do quite well, losing only a small margin. On the other hand, both POMCP variants lose a significant margin to congestion penalties. Box plot 4.12 depicts that SparseQ algorithms as well as Q-learning have no outliers. Oppositely, POMCP has a best performing outlier and POMCP-inf has a worst performing one, but despite that, they achieve good results as the agents' scores are close to the average values. Conversely to the other files' learning curves, Figure 4.10 depicts a smoother learning process for both SparseQ algorithms and especially for SparseQ-agent.

Table 4.6: MCTOPMTW-100-8 results

MCTOPMTW-100-8							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	242	146.48	176.24	270	160	215	7.75
SPARSEQ-Agent	226	148	173.08	260	180	210	7.5
Q-learning	240	180	**207.7**	260	210	236.25	8.75
POMCP-inf	233.6	149.76	176.18	280	170	230	9.125
POMCP	206	148.64	182.82	290	210	**250**	9.25
Random	94	40	64.5	100	40	67.5	5.25

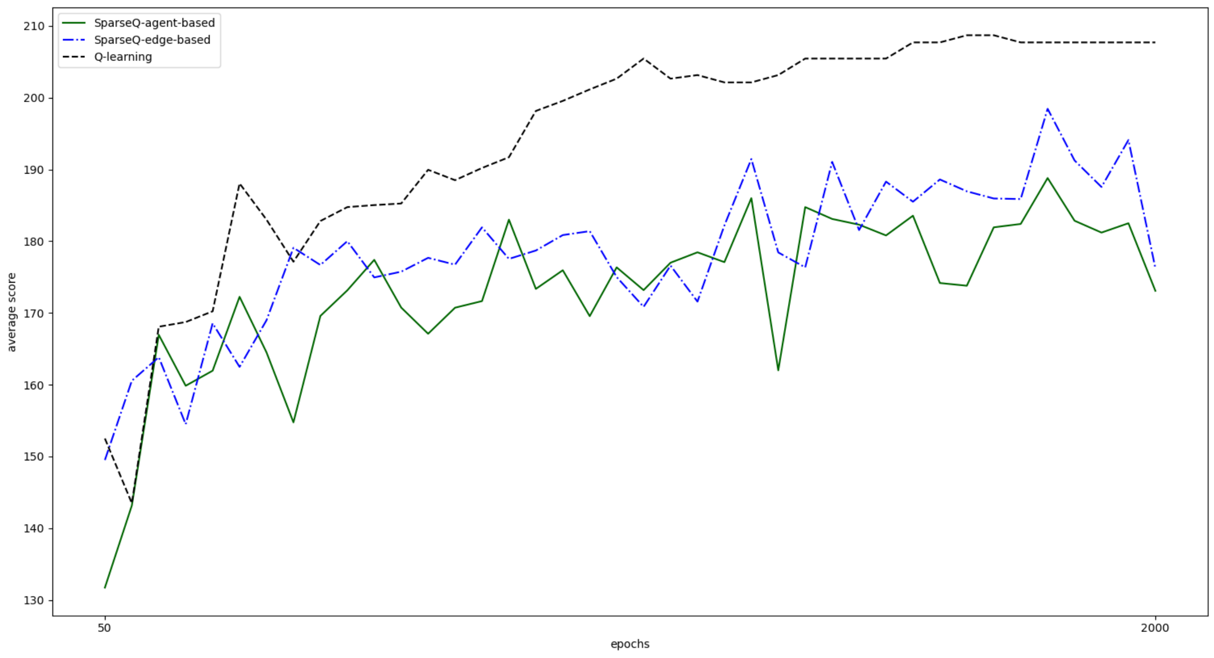
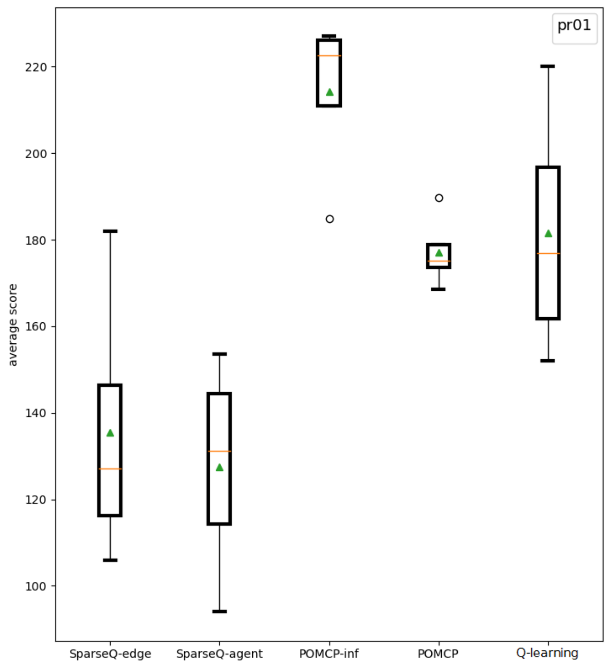


Figure 4.11: Learning curves for MCTOPMTW-100-8 file

Finally, in connection with file MCTOPMTW-100-8 and Table 4.6, Q-learning accomplished the best average score. If not for Q-learning and Random, the remaining algorithms show relatively close performance with POMCP leading by a small margin. Once more, POMCP has the best undiscounted score but conclusively loses a $\frac{182.82}{250} = 23.4\%$ 23.4% of it to congestion, where 182.82 is the average discounted total score while 250 is the undiscounted one (Table 4.6). Concerning the MCTOPMTW-100-8's box plot 4.12, POMCP's minimum score and SparseQ-Agent's maximum score are both outliers, but in spite of that they both show off well distributed values. Also, Q-learning exhibits two outliers.



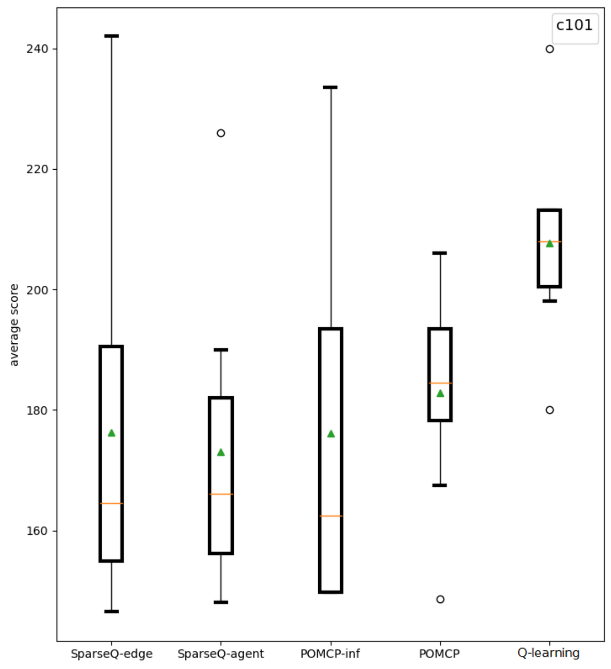


Figure 4.12: Box plots of discounted rewards for MCTOPMTW-48-4 and MCTOPMTW-100-8

In general, POMCP, POMCP-inf and Q-learning rank high in all problem instances. Actually, non-informed POMCP seems to perform better than POMCP-inf in the files when 8 agents were used; this must be due to the heuristic's influence in larger problems. Most certainly, if the heuristic's parameters were fine tuned by hand for each dataset, along with the number of top actions provided, POMCP would attain even higher scores.

Concerning SparseQ-edge and SparseQ-agent, they seem to perform almost the same, with the former being slightly better in most cases. The discrepancies shown in the learning curves are mainly due to the Max-Plus algorithm not being always able to find the best action, mainly because of the large number of joint actions and states, and secondly because of the not-complete CGs used.

Actually, Q-learning performs very well in all datafiles, in respect to the average scores, as well as congestion control and smoothness during training. The only negative aspect is the existence of outliers as shown in the respective box plots that are more common in Q-learning than other algorithms. Despite each agent's negligence of other agents and their impact on its rewards, each one is able to find a good path eventually through exploration. The single agent problem is also much simpler as the state and action spaces are significantly smaller than the multiagent one. This size of the problem from a multiagent perspective is troublesome for SparseQ and we address it in Section 4.4 where we address these limitations and propose solutions.

4.3 Applying Improvements

Here, we detail changes and heuristics used to obtain further improvements in our results as first mentioned in Section 3.5.

Specifically, regarding the heuristic from Section 3.3, the parameters used for each type of OP are the same as those mentioned in Section 4.1. For the TDOP datasets we use only the distance of each node from the last node visited, as well as each available node's score as metrics. Giving:

$$WR(i) = a * SR(i) + b * DSR(i)$$

with:

$$a = \frac{|SR|}{|SR| + |DSR|}$$

$$b = \frac{|DSR|}{|SR| + |DSR|}$$

for each node i where DSR is the distance ranking. Despite the evident lack of information, and the fact that we deemed this inappropriate for POMCP due to poor performance, SparseQ algorithms and Q-learning achieved good results. Finally, we run all the tests again specifically for SparseQ-agent, SparseQ-edge and Q-learning, with the said adjustments implemented. The parameters used are an exponentially decayed ϵ from 1 to 0.05 is used, along with a learning rate α decaying from 1 to 0.1, and a learning factor γ fixed at 0.9. We run 20000 training episodes, and run one fully exploitative episode every 50 ones (which results in 20400 episodes), in order to monitor the performance. Nonetheless, for the plots to be clearer we use the average score every 100 episodes starting from episode 50. We also add previous results by Q-learning, in the learning curves, as a reference point. These results are depicted as a red line which reflects the performance achieved by the end of training. The CGs used for 4 and 5 agents can be seen at Figures 4.13 and 4.14. It is clear that these graphs are denser than before, having 2.5 and 2.8 average branching factors respectively. For reference the previous CGs's corresponding branching factors were 2 and 2.5. On top of that, we ensure this time that each node not only has at most 3 neighbors, but also has at least 2. The new results are portrayed and discussed in Section 4.4.

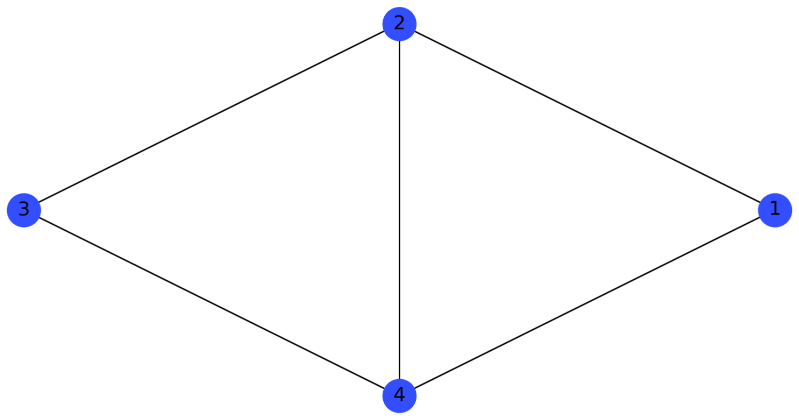


Figure 4.13: New Coordination Graph of 4 agents

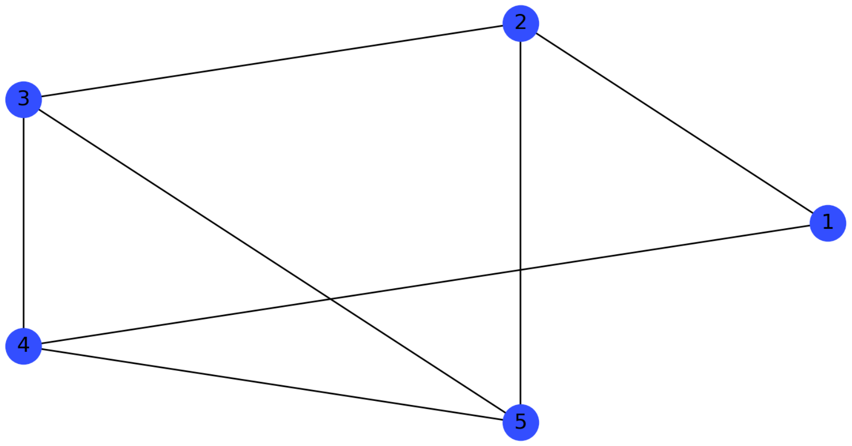


Figure 4.14: New Coordination Graph of 5 agents

4.4 Results with Improved Heuristics and New Settings

Here, we present and discuss the results achieved by SparseQ-agent, SparseQ-edge and Q-learning, after applying the changes analyzed in Section 4.3.

Table 4.7: TOP-66-5 new results

TOP-66-5 - new results							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	466	426	**443.2**	485	445	**466**	17.4
SPARSEQ-Agent	464	348	413.8	485	360	425	16.8
Q-learning	460	405	429	460	405	429	16.4

Starting with the TOP files, for file TOP-66-5 SparseQ-edge achieves the best average score, followed by Q-learning and then SparseQ-agent, even though the differences are rather small. In Figure 4.15 we can see that SparseQ-agent reaches and converges to its peak very early on and that Q-learning shows divergent behavior on the first half of episodes but improves in the second half rapidly and surpasses SparseQ-agent. It is also noteworthy that Q-learning achieves zero congestion by the end of the training. As emerges from the box plot in Figure 4.17, SparseQ-edge has the best quality of rewards, even though there exists an outlier. Q-learning performs very well too, and SparseQ-agent, despite not having any outliers, has more spread rewards among the agents.

Similarly, in the second TOP file, TOP-102-8 SparseQ-edge performs best with the other two falling behind. As shown in Figure 4.16, SparseQ-agent converges

Table 4.8: TOP-102-8 new results

TOP-102-8 - new results							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	189.4	143.24	**164.76**	196	158	**182.25**	8.875
SPARSEQ-Agent	162.48	109.21	133.76	193	155	173.12	8.125
Q-learning	161.84	133.44	145.38	191	141	173	8.75

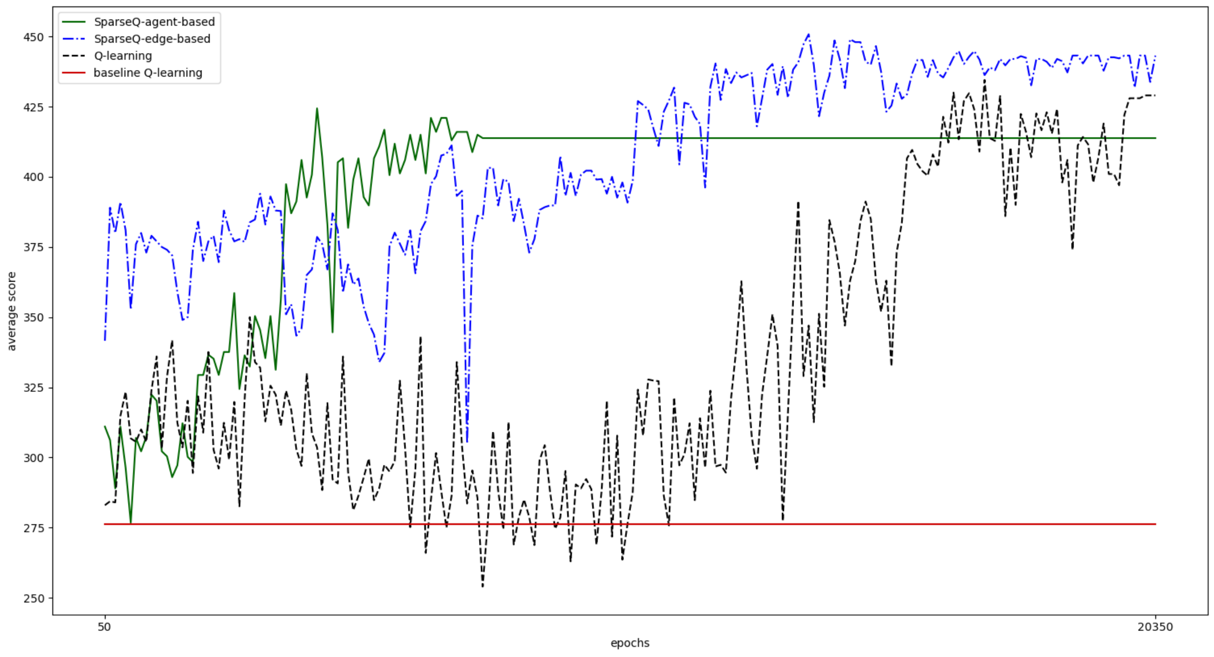


Figure 4.15: New learning curves for TOP-66-5 file

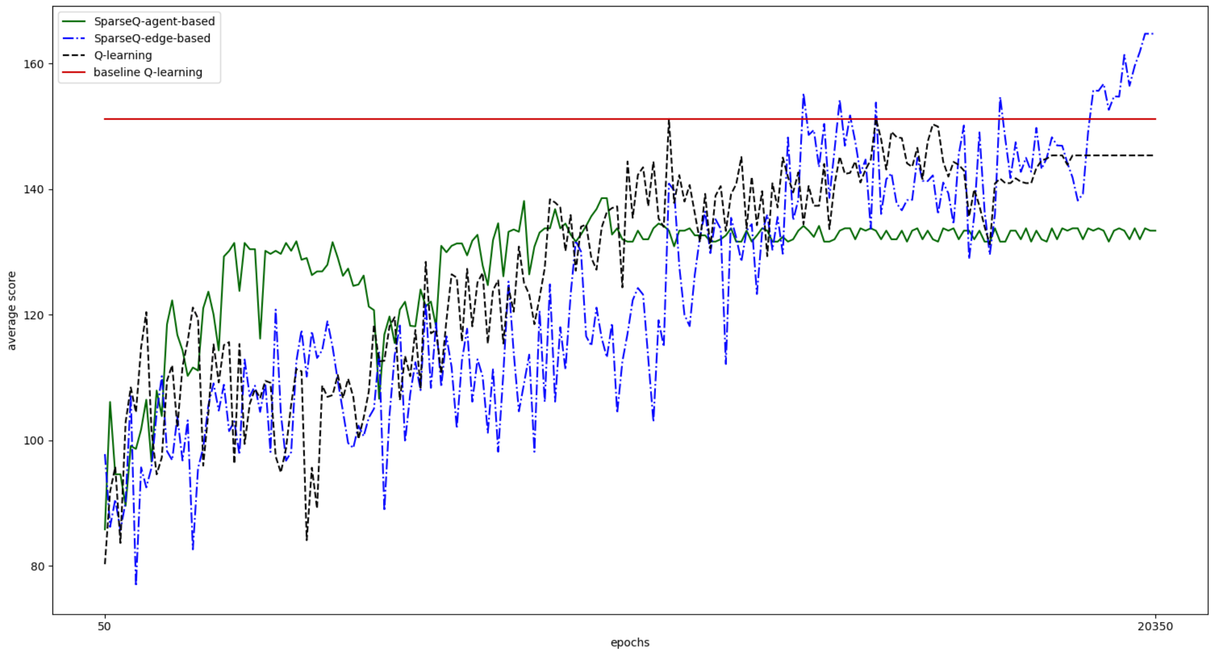


Figure 4.16: New learning curves for TOP-102-8

really fast while Q-learning is more stable in this one. From TOP-102-8's box plot in Figure 4.17 we can conclude that all algorithms do well in terms of fairness among

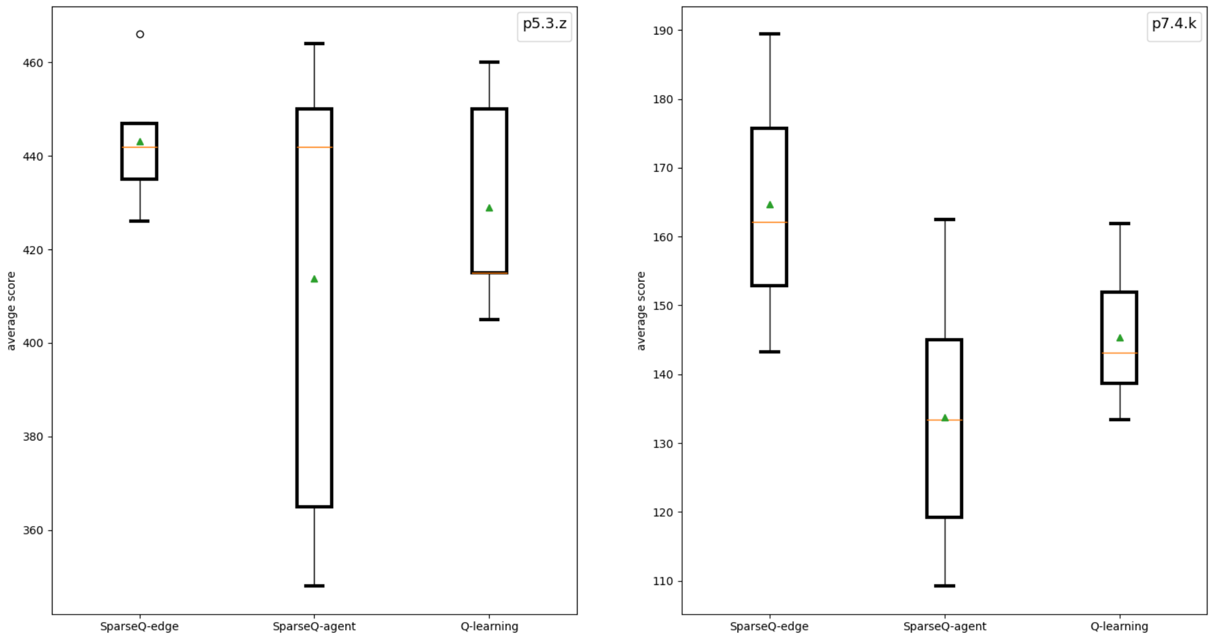


Figure 4.17: New box plots of discounted rewards for TOP-66-5 and TOP-102-8

Table 4.9: TDOP-50-4 new results

TDOP-50-4 - new results							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	205.6	175.6	188.7	213	183	196	6.75
SPARSEQ-Agent	207.8	176.8	**193.2**	215	184	**200.5**	7
Q-learning	207.8	167.6	179.95	215	175	187.25	6.5

the agents, with SparseQ-agent doing a little worse than the other two.

In dataset TDOP-50-4, SparseQ-agent is prevailing. SparseQ-edge follows in close distance and Q-learning lags behind. On top of that, SparseQ-agent is again more stable, with SparseQ-edge being the most unstable algorithm, as evident in Figure 4.18. From Figure 4.20, we get that all algorithms do well enough, with Q-learning being the worst from this perspective.

Table 4.10: TDOP-100-8 new results

TDOP-100-8 - new results							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	253	182.6	212.95	267	199	231.75	8
SPARSEQ-Agent	229.64	179.2	204.09	275	217	**249.25**	8.375
Q-learning	260.8	187.2	**220.59**	282	201	247.75	8.25

Regarding TDOP-100-8, all algorithms achieve very similar performance as is obvious from the respective table and Figure 4.19. It is notable that Q-learning achieves both the highest maximum score and the highest minimum score. The only

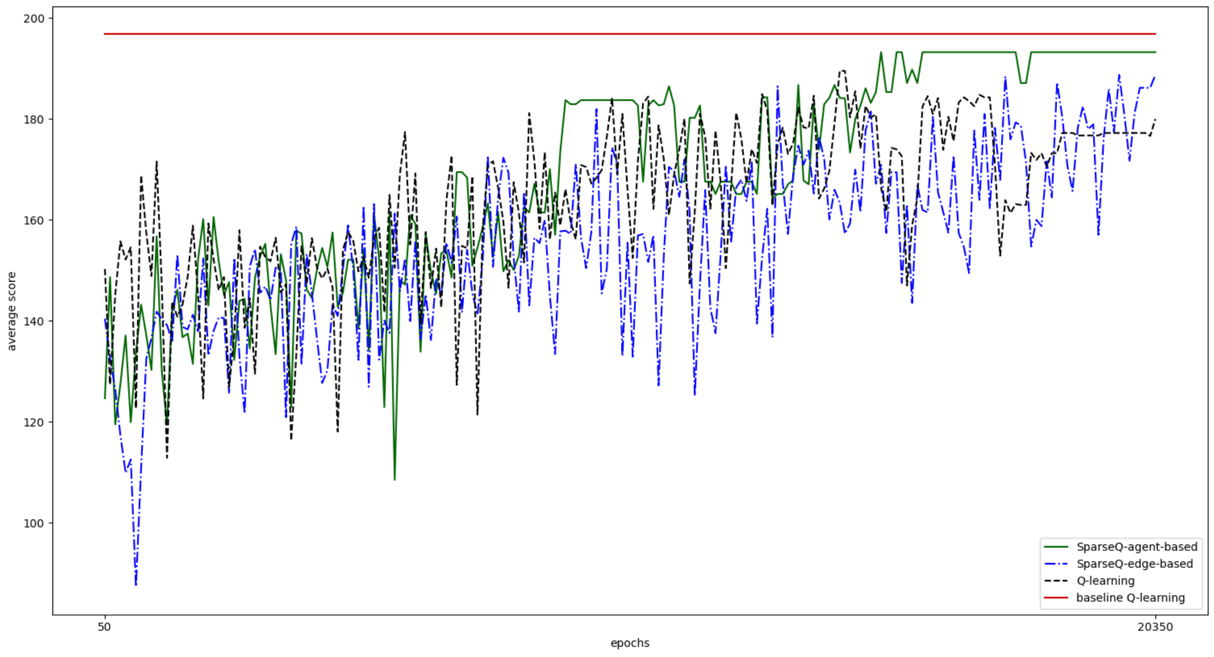


Figure 4.18: New learning curves for TDOP-50-4 file

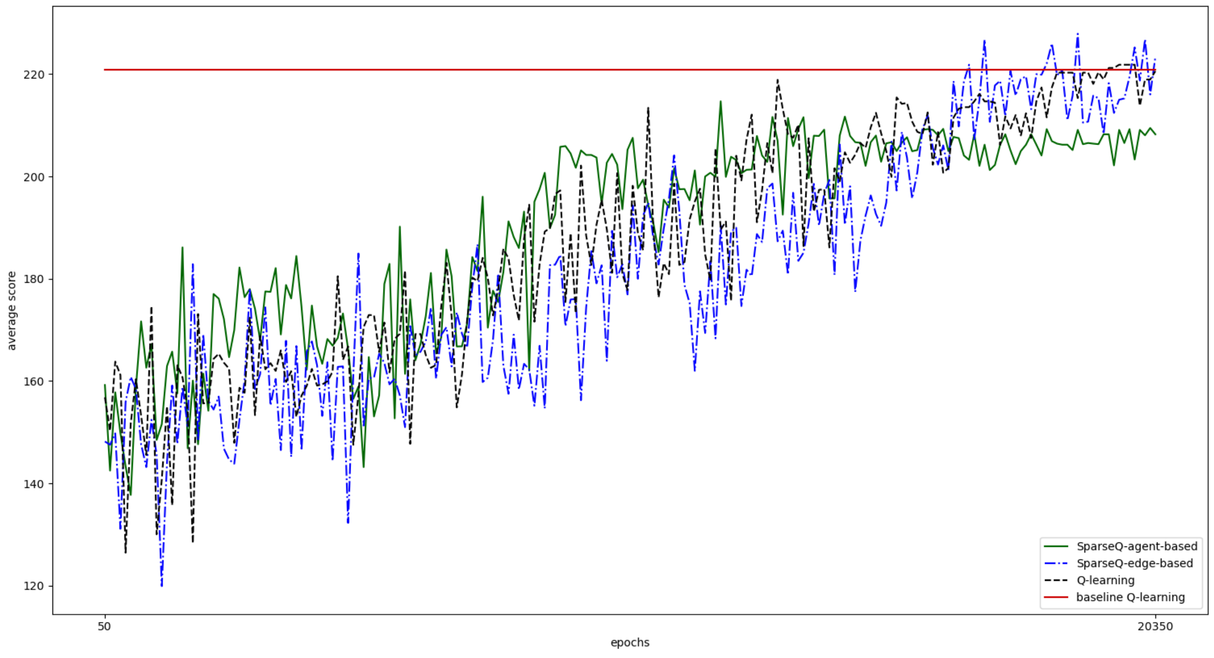


Figure 4.19: New learning curves for TDOP-100-8

algorithm having an outlier is SparseQ-edge (Figure 4.20).

Considering file MCTOPMTW-48-4, Q-learning is the best performing agent

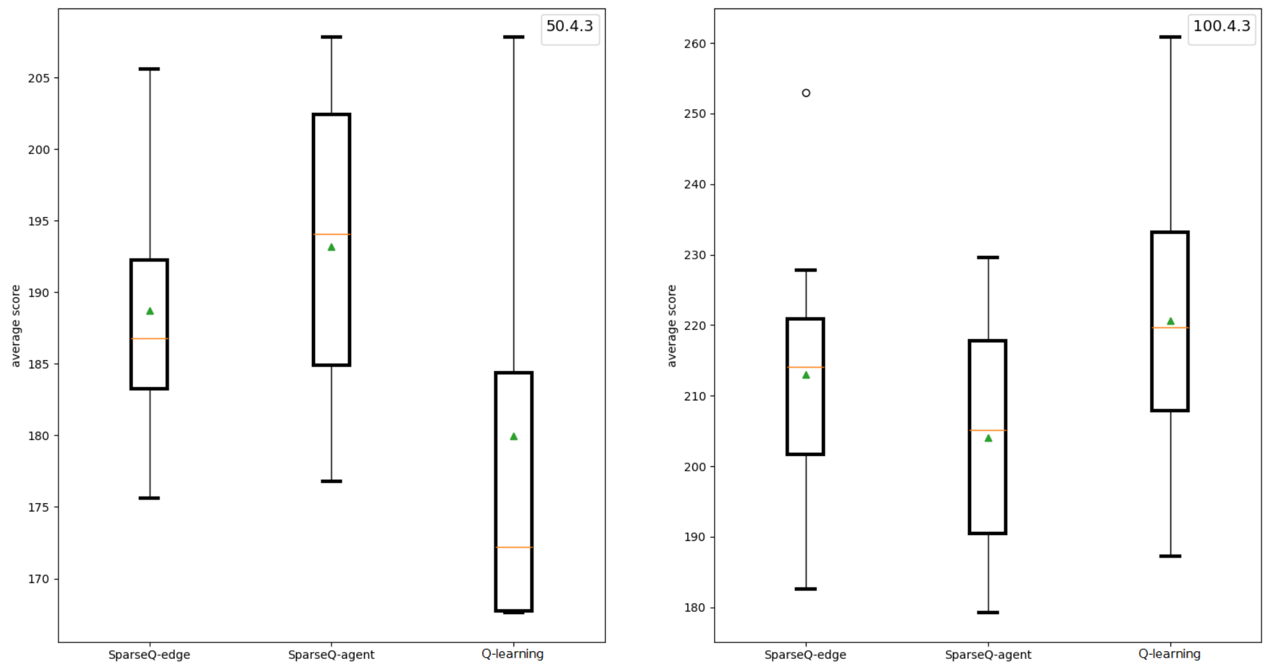


Figure 4.20: New box plots of discounted rewards for TDOP-50-4 and TDOP-100-8

Table 4.11: MCTOPMTW-48-4 new results

MCTOPMTW-48-4 - new results							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	219.2	181	197.6	231	181	203.5	11.75
SPARSEQ-Agent	191.12	134	164.14	200	134	169.75	10.5
Q-learning	225.2	211.39	**217.6**	241	221	**228**	14

followed by SparseQ-edge and then SparseQ-agent. This time, Q-learning does significantly more steps by average than the other two algorithms, and also achieves a very high minimum score, it is remarkable that the best performing SparseQ-agent agent does worse than the worst performing Q-learning one. The learning curves (Figure 4.21) appear to be the smoothest ones so far. Q-learning does extremely well in the uniformity of the scores too, as shown in Figure 4.23.

Table 4.12: MCTOPMTW-100-8 new results

MCTOPMTW-100-8 - new results							
	congestion discounts applied			**congestion discounts not applied**			
Algorithm	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Steps**
SPARSEQ-Edge	240	160	**196.3**	250	160	227.5	8.875
SPARSEQ-Agent	189.6	125.84	152.51	230	160	211.25	7.75
Q-learning	230	168	193.2	240	220	**230**	9.375

Finally, we have have new results for dataset MCTOPMTW-100-8. In this problem instance, Q-learning does very well but slightly worse than SparseQ-edge.

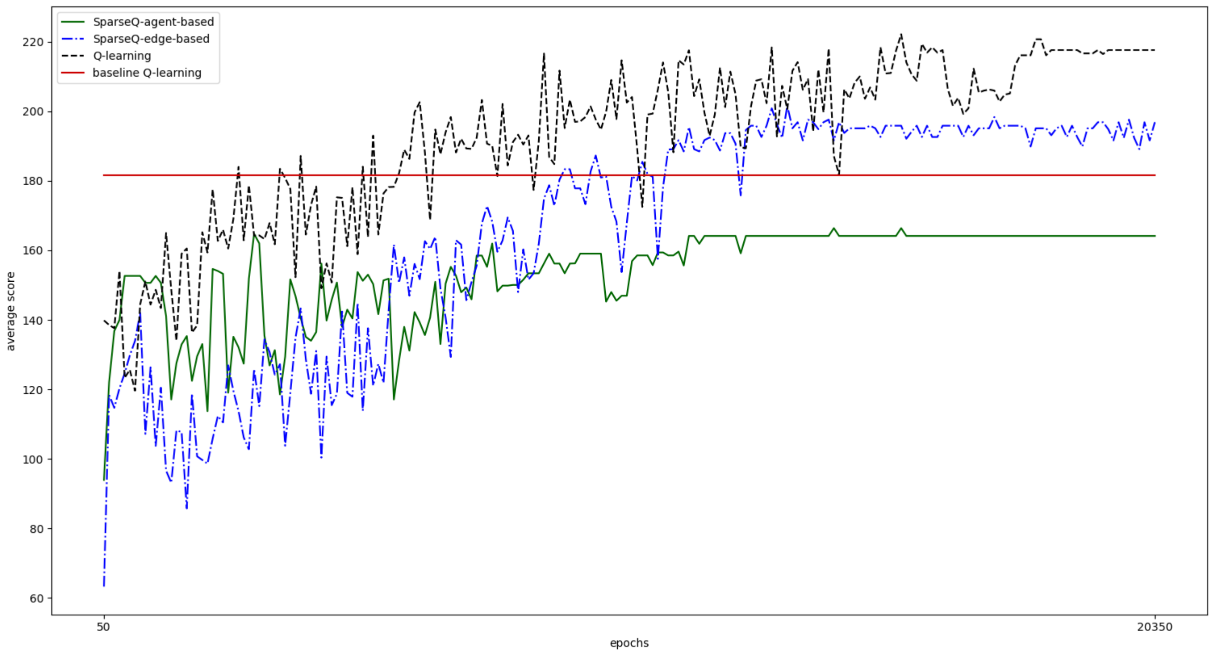


Figure 4.21: New learning curves for MCTOPMTW-48-4 file

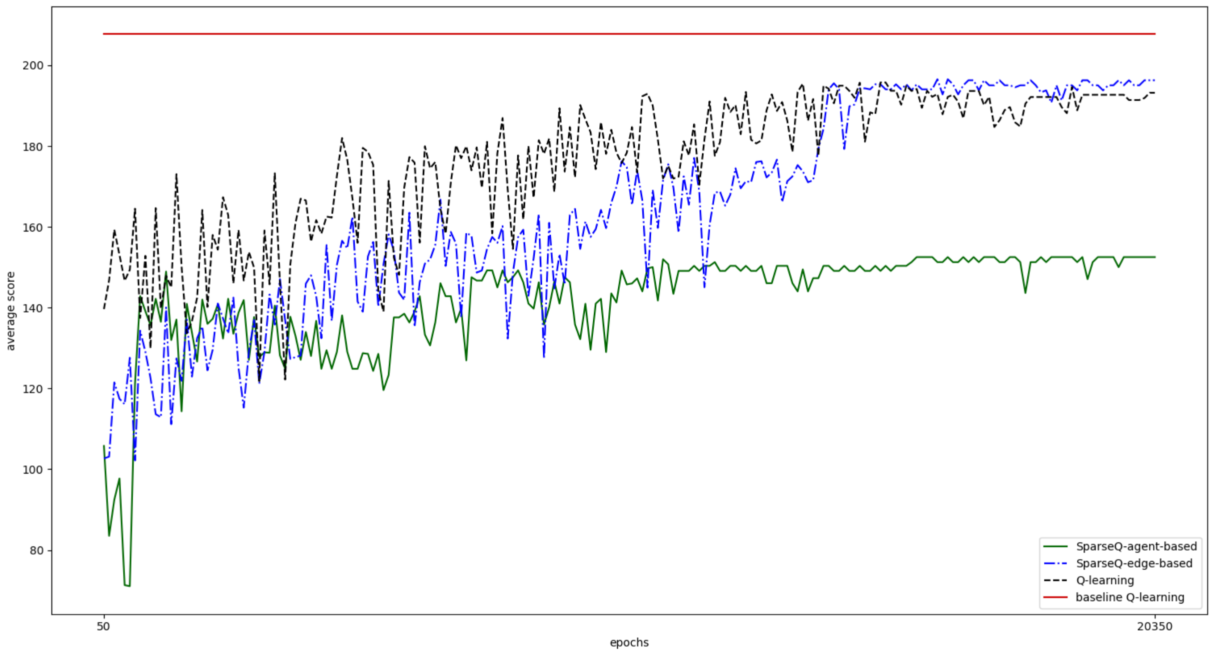


Figure 4.22: New learning curves for MCTOPMTW-100-8

On the other hand, SparseQ-agent exhibits a disappointing performance. But, apparently all of them have rather smooth learning curves in comparison to previous

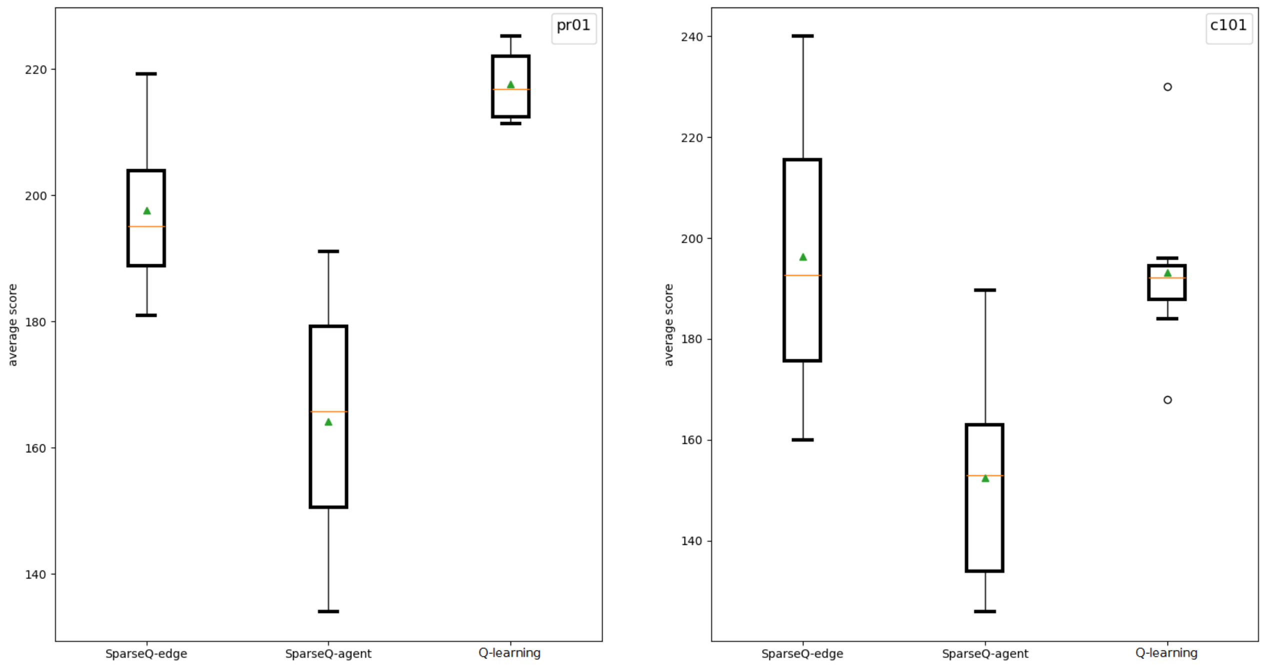


Figure 4.23: New box plots of discounted rewards for MCTOPMTW-48-4 and MCTOPMTW-100-8

ones (Figure 4.22). Unfortunately, Figure 4.23 depicts the existence of two outliers for Q-learning, and one of them being an underperforming one.

Generally, the changes caused a significant boost in the performance of SparseQ-edge and SparseQ-agent. The only instance where we witness a reduction in score is for SparseQ-agent in the MCTOPMTW-100-8 file. Q-learning improved its results in half of the files, but then again, the changes were targeted mainly for SparseQ. Since the number of available actions is significantly reduced, Q-learning finds more difficulty in finding the best paths, as it does not use cooperation or modeling of the other agents' impact. Generally, the performance of these algorithms relies notably on the efficiency of the heuristic, along with the choice of its parameters, in finding the best actions. This can explain why the baseline was not surpassed in some of the datasets.

Chapter 5

Conclusions and Future Work

In this thesis we introduce a new extension to the OP, viewing it as a multi-agent environment and imposing congestion-driven discounts on the rewards. Furthermore, we modeled the problem as both a multi-agent MDP and a POMDP, and solved it using AI methods, like SparseQ and POMCP. Six algorithms were used in total, namely SparseQ with edge decomposition and both agent and edge based updates, classic POMCP and a version of POMCP utilizing problem specific information, Q-learning, and one performing uniformly random actions.

These algorithms were tested on six different problem instances of three different types, OPTW, TOP, and MCTOPTW. Due to the complexity of these problems, this proves to be a non-trivial task, but some of the algorithms tested give promising results. In particular, the large size of the state space and the action space, as initially defined, was a challenging issue especially for the Q-learning variants. To mitigate this problem, a relaxation to the action set was employed along with the simplification of the state representation, drastically improved the performance. We also used the heuristic introduced in [Golden et al., 1987] both in the informed version of POMCP and in SparseQ and Q-learning algorithms. It would definitely prove beneficial to consider a smarter way to select the heuristic's parameters or, at least, fine tune them by hand.

Another promising idea is to embed rollouts in SparseQ and Q-learning. More specifically, we could run a few rollouts for each state-action pair in order to gain a good initial estimate for the respective Q-value. As expected, this technique would cause the training process to be much slower, but we foresee that it could achieve good results with fewer episodes. It would prove interesting to test these algorithms in more challenging problems, with a much higher agents to node ratio. Additionally, closely related problems can be taken into consideration, such as the Vehicle Routing Problem (VRP) [Gunawan et al., 2019], the Selfish Orienteering Problem (SOP) [Varakantham et al., 2015] and the Multi-Agent Orienteering Problem with Time Capacity Constraints (MOPTCC) [Chen et al., 2014]. It would be interesting to study how this work compares against known methods for these problem variants.

Looking into further extensions, a technique from a recent work using a policy gradient algorithm, namely Stochastic Action Set Policy Gradient (SAS-PG) [Chandak et al., 2020] could be proven promising, especially since it has been tested with success on a routing problem. Finally, another popular solver, Determinized Sparse Partially Observable Tree (DESPOT) [Somani et al., 2013], could be tried and compared to POMCP. As a matter of fact, the two algorithms have many similarities, with DESPOT using a belief tree with action and observation nodes but evaluating policies using sampled scenarios.

Bibliography

Archetti, C., Bianchessi, N., Speranza, M. G., & Hertz, A. (2014). The split delivery capacitated team orienteering problem. *Networks*, *63*(1), 16–33.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2-3), 235–256.

Best, G., Cliff, O. M., Patten, T., Mettu, R. R., & Fitch, R. (2020). Decentralised Monte Carlo tree search for active perception. *Algorithmic Foundations of Robotics XII* (pp. 864–879). Springer.

Boutilier, C., Cohen, A., Daniely, A., Hassidim, A., Mansour, Y., Meshi, O., Mladenov, M., & Schuurmans, D. (2018). Planning and learning with stochastic action sets. *arXiv preprint arXiv:1805.02363*.

Brown, G. W. (1951). Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, *13*(1), 374–376.

Castellini, A., Chalkiadakis, G., & Farinelli, A. (2019). Influence of state-variable constraints on partially observable monte carlo planning. *Proceedings of 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 5540–5546.

Chandak, Y., Theocharous, G., Metevier, B., & Thomas, P. S. (2020). Reinforcement Learning When All Actions Are Not Always Available. *AAAI*, 3381–3388.

Chao, I.-M. (1993). *Algorithms and solutions to multi-level vehicle routing problems*. University of Maryland at College Park.

Chao, I.-M., Golden, B. L., & Wasil, E. A. (1996). The team orienteering problem. *European journal of operational research*, *88*(3), 464–474.

Chen, C., Cheng, S.-F., & Lau, H. C. (2014). Multi-agent orienteering problem with time-dependent capacity constraints. *Web Intelligence and Agent Systems: An International Journal*, *12*(4), 347–358.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. *International conference on computers and games*, 72–83.

Fomin, F. V., & Lingas, A. (2002). Approximation algorithms for time-dependent orienteering. *Information Processing Letters*, *83*(2), 57–62.

Gama, R., & Fernandes, H. L. (2020). A Reinforcement Learning Approach to the Orienteering Problem with Time Windows. *arXiv preprint arXiv:2011.03647*.

Gavalas, D., Konstantopoulos, C., Mastakas, K., & Pantziou, G. (2014). A survey on algorithmic approaches for solving tourist trip design problems. *Journal of Heuristics*, *20*(3), 291–328.

Gavalas, D., Konstantopoulos, C., Mastakas, K., Pantziou, G., & Vathis, N. (2015). Heuristics for the time dependent team orienteering problem: Application to tourist route planning. *Computers & Operations Research*, *62*, 36–50.

Golden, B. L., Levy, L., & Vohra, R. (1987). The orienteering problem. *Naval Research Logistics (NRL)*, *34*(3), 307–318.

Guestrin, C. (2003). *Planning under uncertainty in complex structured environments* (Doctoral dissertation). Stanford University.

Guestrin, C., Koller, D., & Parr, R. (2002). Multiagent planning with factored MDPs. *Advances in neural information processing systems*, 1523–1530.

Gunawan, A., Lau, H. C., & Lu, K. (2015). An iterated local search algorithm for solving the orienteering problem with time windows. *European conference on evolutionary computation in combinatorial optimization*, 61–73.

Gunawan, A., Lau, H. C., & Vansteenwegen, P. (2016). Orienteering problem: A survey of recent variants, solution approaches and applications. *European Journal of Operational Research*, *255*(2), 315–332.

Gunawan, A., Vincent, F. Y., Widjaja, A. T., & Vansteenwegen, P. (2019). Simulated Annealing for the Multi-Vehicle Cyclic Inventory Routing Problem. *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, 691–696.

Hoffman, K. L., Padberg, M., Rinaldi, G., et al. (2013). Traveling salesman problem. *Encyclopedia of operations research and management science*, *1*, 1573–1578.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, *101*(1-2), 99–134.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, *4*, 237–285.

Kantor, M. G., & Rosenwein, M. B. (1992). The orienteering problem with time windows. *Journal of the Operational Research Society*, *43*(6), 629–635.

Ke, L., Archetti, C., & Feng, Z. (2008). Ants can solve the team orienteering problem. *Computers & Industrial Engineering*, *54*(3), 648–665.

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *European conference on machine learning*, 282–293.

Kok, J. R., & Vlassis, N. (2006). Collaborative multiagent reinforcement learning by payoff propagation. *Journal of Machine Learning Research*, *7*(Sep), 1789–1828.

Lambert Iii, T. J., Epelman, M. A., & Smith, R. L. (2005). A fictitious play approach to large-scale optimization. *Operations Research*, *53*(3), 477–489.

Le, T. V., Liu, S., & Lau, H. C. (2016). A Reinforcement Learning Framework for Trajectory Prediction Under Uncertainty and Budget Constraint. *ECAI*, 347–354.

Li, Y. (2018). Queuing theory with heavy tails and network traffic modeling.

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

Orienteering. (2020). Orienteering — Wikipedia, The Free Encyclopedia [[Online; accessed 19-January-2021]]. https://en.wikipedia.org/w/index.php?title=Orienteering&oldid=989257992

Paxson, V., & Floyd, S. (1994). Wide-area traffic: the failure of Poisson modeling. *ACM SIGCOMM Computer Communication Review*, *24*(4), 257–268.

Pěnička, R., Faigl, J., & Saska, M. (2019). Physical orienteering problem for unmanned aerial vehicle data collection planning in environments with obstacles. *IEEE Robotics and Automation Letters*, *4*(3), 3005–3012.

Poupart, P. (2005). *Exploiting structure to efficiently solve large scale partially observable Markov decision processes.* Citeseer.

Sevkli, Z., & Sevilgen, F. E. (2006). Variable neighborhood search for the orienteering problem. *International Symposium on Computer and Information Sciences*, 134–143.

Silver, D., & Veness, J. (2010). Monte-Carlo planning in large POMDPs.

Somani, A., Ye, N., Hsu, D., & Lee, W. S. (2013). DESPOT: Online POMDP planning with regularization. *Advances in neural information processing systems*, 1772–1780.

Souffriau, W. (2010). Automated tourist decision support. *Katholieke Universiteit Leuven, Mar.*

Souffriau, W., Vansteenwegen, P., Vanden Berghe, G., & Van Oudheusden, D. (2013). The multiconstraint team orienteering problem with multiple time windows. *Transportation Science*, *47*(1), 53–63.

Sutton, R. S., & Barto, m., Andrew G. (2018). *Reinforcement learning: An introduction.*

Thrun, S. (2000). Monte carlo pomdps. *Advances in neural information processing systems*, 1064–1070.

Tsiligirides, T. (1984). Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, *35*(9), 797–809.

Vansteenwegen, P., Souffriau, W., & Van Oudheusden, D. (2011). The orienteering problem: A survey. *European Journal of Operational Research*, *209*(1), 1–10.

Varakantham, P., Mostafa, H., Fu, N., & Lau, H. C. (2015). DIRECT: A scalable approach for route guidance in selfish orienteering problems.

Verbeeck, C., Vansteenwegen, P., & Aghezzaf, E.-H. (2016). Solving the stochastic time-dependent orienteering problem with time windows. *European Journal of Operational Research*, *255*(3), 699–718.

Verbeeck, C., Vansteenwegen, P., & Aghezzaf, E.-H. (2017a). The Time-Dependent Orienteering Problem with Time Windows: A Fast Ant Colony System. *Annals of Operations Research*, *254*(1-2), 481–505. $$Uhttps://lirias.kuleuven.be/retrieve/430946$$DTDOPTW_final_preprint.pdf%20[freely%20available]

Verbeeck, C., Vansteenwegen, P., & Aghezzaf, E.-H. (2017b). The time-dependent orienteering problem with time windows: a fast ant colony system. *Annals of Operations Research*, *254*(1), 481–505.

Vlassis, N. (2007). A concise introduction to multiagent systems and distributed artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *1*(1), 1–71.

Walraven, E., Spaan, M. T., & Bakker, B. (2016). Traffic flow optimization: A reinforcement learning approach. *Engineering Applications of Artificial Intelligence*, *52*, 203–212.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Yu, J., Schwager, M., & Rus, D. (2016). Correlated orienteering problem and its application to persistent monitoring tasks. *IEEE Transactions on Robotics*, *32*(5), 1106–1118.

Appendices

Appendix A

Instance TOP-66-5

```
66
m 3
tmax 43.3
-0.500 0.000 0
-7.000 -7.000 35
-7.000 -5.000 35
-7.000 -3.000 35
-7.000 -1.000 35
-7.000 1.000 35
-7.000 3.000 35
-7.000 5.000 35
-7.000 7.000 35
-5.000 -7.000 35
-5.000 -5.000 25
-5.000 -3.000 25
-5.000 -1.000 25
-5.000 1.000 25
-5.000 3.000 25
-5.000 5.000 25
-5.000 7.000 35
-3.000 -7.000 35
-3.000 -5.000 25
-3.000 -3.000 15
-3.000 -1.000 15
-3.000 1.000 15
-3.000 3.000 15
-3.000 5.000 25
-3.000 7.000 35
-1.000 -7.000 35
-1.000 -5.000 25
-1.000 -3.000 15
-1.000 -1.000 5
-1.000 1.000 5
-1.000 3.000 15
-1.000 5.000 25
```

```
-1.000 7.000 35
1.000 -7.000 35
1.000 -5.000 25
1.000 -3.000 15
1.000 -1.000 5
1.000 1.000 5
1.000 3.000 15
1.000 5.000 25
1.000 7.000 35
3.000 -7.000 35
3.000 -5.000 25
3.000 -3.000 15
3.000 -1.000 15
3.000 1.000 15
3.000 3.000 15
3.000 5.000 25
3.000 7.000 35
5.000 -7.000 35
5.000 -5.000 25
5.000 -3.000 25
5.000 -1.000 25
5.000 1.000 25
5.000 3.000 25
5.000 5.000 25
5.000 7.000 35
7.000 -7.000 35
7.000 -5.000 35
7.000 -3.000 35
7.000 -1.000 35
7.000 1.000 35
7.000 3.000 35
7.000 5.000 35
7.000 7.000 35
0.500 0.000 0
```

Appendix B

Instance TOP-102-8

```
102
m 4
tmax 55.0
35.000 35.000 0
15.000 30.000 26
55.000 5.000 29
31.000 52.000 27
60.000 12.000 31
8.000 56.000 27
13.000 52.000 36
6.000 68.000 30
21.000 24.000 28
56.000 39.000 36
55.000 54.000 26
16.000 22.000 41
4.000 18.000 35
28.000 18.000 26
26.000 27.000 27
55.000 45.000 13
55.000 20.000 19
55.000 60.000 16
30.000 60.000 16
20.000 65.000 12
50.000 35.000 19
30.000 25.000 23
15.000 10.000 20
10.000 20.000 19
20.000 40.000 12
15.000 60.000 17
45.000 20.000 11
45.000 10.000 18
45.000 30.000 17
35.000 40.000 16
41.000 37.000 16
40.000 60.000 21
```

```
35.000 69.000 23
53.000 52.000 11
65.000 55.000 14
5.000 5.000 16
11.000 14.000 18
6.000 38.000 16
47.000 47.000 13
37.000 31.000 14
57.000 29.000 18
36.000 26.000 18
12.000 24.000 13
24.000 58.000 19
62.000 77.000 20
49.000 73.000 25
67.000 5.000 25
57.000 68.000 15
47.000 16.000 25
49.000 11.000 18
49.000 42.000 13
53.000 43.000 14
57.000 48.000 23
15.000 47.000 16
14.000 37.000 11
26.000 35.000 15
18.000 24.000 22
25.000 24.000 20
22.000 27.000 11
25.000 21.000 12
18.000 18.000 17
41.000 49.000 10
35.000 17.000 7
25.000 30.000 3
20.000 50.000 5
10.000 43.000 9
30.000 5.000 8
5.000 30.000 2
45.000 65.000 9
65.000 35.000 3
65.000 20.000 6
64.000 42.000 9
63.000 65.000 8
2.000 60.000 5
20.000 20.000 8
40.000 25.000 9
42.000 7.000 5
24.000 12.000 5
23.000 3.000 7
2.000 48.000 1
```

```
49.000 58.000 10
27.000 43.000 9
63.000 23.000 2
53.000 12.000 6
32.000 12.000 7
17.000 34.000 3
27.000 69.000 10
15.000 77.000 9
37.000 47.000 6
37.000 56.000 5
44.000 17.000 9
46.000 13.000 8
61.000 52.000 3
56.000 37.000 6
11.000 31.000 7
26.000 52.000 9
31.000 67.000 3
15.000 19.000 1
22.000 22.000 2
19.000 21.000 10
20.000 26.000 9
35.000 35.000 0
```

Appendix C

Instance TDOP-50-4

```
50
50400000
11174;0;0;0;21600000;72000000
10467;33;7;1152295;25200000;64800000
21660;7;2;1440728;50400000;68400000
26440;16;4;1192523;43200000;72000000
20025;25;5;1149992;54000000;68400000
29511;27;6;858459;43200000;61200000
20650;5;1;820593;39600000;54000000
8314;29;6;1524209;36000000;57600000
28023;24;5;1391980;32400000;50400000
14019;29;6;578310;50400000;72000000
9906;19;4;2027233;25200000;43200000
7392;24;5;1185928;28800000;72000000
3626;36;8;1669281;46800000;68400000
4415;33;7;1494345;28800000;72000000
25825;7;2;991878;43200000;61200000
25875;37;8;1666536;25200000;57600000
20160;17;4;1329891;46800000;61200000
28071;39;8;1333473;50400000;64800000
28298;31;7;1280576;43200000;72000000
8178;29;6;1334655;54000000;68400000
32271;9;2;1139862;36000000;72000000
2669;33;7;1220866;28800000;64800000
13986;22;5;1451646;32400000;46800000
8481;3;1;1257706;32400000;46800000
7628;6;2;1363468;46800000;61200000
4100;30;6;1427142;43200000;68400000
2626;23;5;1532733;21600000;57600000
1925;26;6;1491803;25200000;72000000
29973;36;8;845117;39600000;72000000
14182;38;8;1192548;43200000;68400000
27433;34;7;1063102;25200000;57600000
27594;28;6;676419;32400000;54000000
13032;30;6;1073103;21600000;72000000
```

```
143;24;5;606060;25200000;64800000
31287;39;8;1257933;54000000;68400000
7901;39;8;1356217;32400000;50400000
8361;28;6;989770;28800000;43200000
30975;36;8;913928;25200000;46800000
29171;2;1;993832;43200000;61200000
30834;17;4;1065414;46800000;68400000
25761;21;5;895233;25200000;57600000
4668;32;7;938833;28800000;72000000
12551;24;5;1255188;54000000;68400000
13695;5;1;837167;39600000;72000000
21625;18;4;1310760;50400000;68400000
2126;18;4;1791430;43200000;68400000
21695;39;8;1792285;50400000;68400000
26303;17;4;1476167;25200000;61200000
22467;38;8;1370093;43200000;64800000
22594;0;39;0;21600000;72000000
```

Appendix D

Instance TDOP-100-8

```
100
50400000
27466;0;0;0;21600000;72000000
20268;8;2;1434809;43200000;61200000
19794;32;7;1592089;28800000;43200000
25473;30;6;1260918;32400000;57600000
22831;20;4;1048901;46800000;72000000
28443;30;6;1399138;39600000;57600000
13878;31;7;1298356;36000000;68400000
703;13;3;1157143;54000000;72000000
1382;16;4;1453688;50400000;72000000
8824;15;3;1174585;28800000;68400000
8024;5;1;1405310;46800000;64800000
16596;7;2;1063828;28800000;72000000
2328;31;7;1435319;50400000;68400000
31311;25;5;1506397;54000000;72000000
11059;10;2;807427;54000000;72000000
9488;33;7;1090922;21600000;36000000
32529;36;8;1738989;43200000;72000000
2259;38;8;1091492;28800000;68400000
9861;5;1;1507890;32400000;64800000
21287;13;3;1667261;25200000;72000000
8611;27;6;1191101;46800000;61200000
7129;27;6;1019728;36000000;72000000
5842;10;2;1702990;25200000;72000000
3504;9;2;1223822;50400000;72000000
24866;19;4;1474324;54000000;68400000
1882;15;3;1181198;54000000;72000000
22751;28;6;886670;43200000;72000000
18599;28;6;980028;46800000;68400000
2662;12;3;1245240;36000000;61200000
32757;21;5;1215390;43200000;57600000
20279;29;6;1339964;25200000;54000000
19436;9;2;927193;54000000;68400000
32076;21;5;893148;50400000;64800000
```

```
1387;34;7;822946;32400000;46800000
8361;32;7;730607;39600000;72000000
26049;37;8;1192444;46800000;72000000
29493;32;7;830070;25200000;50400000
23841;20;4;1551687;32400000;50400000
1736;38;8;1385244;54000000;68400000
11600;21;5;707289;39600000;57600000
21893;3;1;761979;36000000;72000000
7329;25;5;1259619;28800000;61200000
11370;6;2;554509;54000000;68400000
21795;18;4;1285550;21600000;64800000
9253;26;6;1378532;28800000;46800000
17433;20;4;1020818;25200000;57600000
7209;28;6;1067230;43200000;61200000
3498;39;8;1034132;21600000;72000000
27650;20;4;1208531;21600000;72000000
26842;34;7;1476705;50400000;68400000
16101;1;1;1180361;21600000;39600000
30649;8;2;1248326;25200000;64800000
19852;22;5;1070714;28800000;54000000
28634;18;4;1307162;21600000;39600000
27201;4;1;1213339;25200000;72000000
9991;36;8;602382;25200000;39600000
4920;36;8;1115134;32400000;46800000
22579;23;5;944036;32400000;68400000
32545;26;6;1342452;28800000;43200000
13488;3;1;1021752;25200000;68400000
22526;21;5;916790;46800000;64800000
5539;5;1;1532598;43200000;64800000
6194;17;4;975965;43200000;72000000
25012;1;1;1355582;46800000;61200000
15835;8;2;1101671;32400000;57600000
31498;17;4;1200858;21600000;39600000
18530;29;6;1262286;36000000;72000000
18806;10;2;1058187;46800000;64800000
13393;29;6;1275848;50400000;72000000
13550;12;3;1333893;39600000;61200000
26980;6;2;996707;50400000;64800000
9278;35;7;1443596;43200000;68400000
20194;22;5;1263639;25200000;46800000
21498;21;5;924576;43200000;61200000
31277;18;4;1097394;28800000;46800000
6583;7;2;1270437;32400000;68400000
11160;22;5;1207748;28800000;64800000
26490;4;1;1488663;39600000;57600000
3450;16;4;1643735;21600000;36000000
9073;7;2;1109458;50400000;64800000
27009;5;1;1317669;25200000;50400000
```

```
10209;4;1;1668260;50400000;68400000
18504;5;1;1470662;50400000;68400000
32608;1;1;570521;39600000;57600000
12075;34;7;1543070;28800000;50400000
12612;27;6;1233926;50400000;72000000
28762;20;4;1377679;25200000;61200000
12891;3;1;851397;28800000;72000000
16684;28;6;1320914;25200000;50400000
19933;25;5;1561680;50400000;68400000
2742;2;1;1251795;43200000;61200000
6814;16;4;1368552;50400000;64800000
10397;17;4;1285709;32400000;64800000
20616;24;5;1617192;46800000;68400000
2600;13;3;693994;54000000;72000000
4681;34;7;1436845;46800000;72000000
27033;36;8;783771;21600000;43200000
32585;26;6;1167751;28800000;43200000
3518;3;1;1636677;54000000;72000000
8671;0;39;0;21600000;72000000
```

Appendix E

Instance MCTOPMTW-48-4

```
1 48 994 4 10 3 6 4 2 4 6 4 6
0 -10.442 19.999 0.0 0.0 0 1000
2 -29.73 64.136 2.0 12.0 354 392.8 431.5 470.2 509 0 77 0 0 0 0 0 0 0 0 0 0
3 -30.664 5.463 7.0 8.0 234 275.8 317.5 359.2 401 1 77 1 1 0 1 0 0 1 0 0 0
4 51.642 5.469 21.0 16.0 411 451.5 492 532.5 573 1 42 0 0 1 0 0 1 1 1 0 1
5 -13.171 69.336 24.0 5.0 474 511 548 585 622 1 70 0 0 0 0 0 1 0 0 0 0
6 -67.413 68.323 1.0 12.0 155 190 225 260 295 1 75 1 0 0 0 0 0 0 0 0 1
7 48.907 6.274 17.0 5.0 361 398 435 472 509 1 46 0 0 1 0 0 0 0 0 0 0
8 5.243 22.26 6.0 13.0 451 495.5 540 584.5 629 0 50 0 1 1 1 0 0 0 0 1 0
9 -65.002 77.234 5.0 20.0 425 465.8 506.5 547.2 588 1 27 0 0 0 0 0 0 0 1 0 0
10 -4.175 -1.569 7.0 13.0 72 103.8 135.5 167.2 199 1 62 0 0 0 0 0 0 0 0 0 0
11 23.029 11.639 1.0 18.0 157 197.2 237.5 277.8 318 0 79 0 1 1 1 0 0 0 1 0 0
12 25.482 6.287 4.0 7.0 296 333 370 407 444 1 30 0 0 0 1 0 0 1 1 0 1
13 -42.615 -26.392 10.0 6.0 111 145.5 180 214.5 249 0 4 0 0 0 1 1 0 0 0 0 0
14 -76.672 99.341 2.0 9.0 368 408 448 488 528 1 9 0 0 0 0 0 1 0 0 1 1
15 -20.673 57.892 16.0 9.0 98 136.2 174.5 212.8 251 1 25 0 0 0 1 0 0 0 0 0 0
16 -52.039 6.567 23.0 4.0 96 124 152 180 208 1 25 0 0 1 0 1 1 1 1 1 0
17 -41.376 50.824 18.0 25.0 382 420.2 458.5 496.8 535 0 83 0 1 0 0 1 0 0 1 0 1
18 -91.943 27.588 3.0 5.0 436 472 508 544 580 1 10 0 0 1 0 0 0 0 0 0 1
19 -65.118 30.212 15.0 17.0 405 435.8 466.5 497.2 528 1 67 0 1 0 0 0 0 0 0 0 0
20 18.597 96.716 13.0 3.0 255 294.8 334.5 374.2 414 1 65 0 0 0 0 0 0 0 0 0 1
21 -40.942 83.209 10.0 16.0 293 337.2 381.5 425.8 470 1 85 1 0 0 0 0 0 1 0 0 0
22 -37.756 -33.325 4.0 25.0 298 325.5 353 380.5 408 1 41 0 0 0 0 0 0 1 0 0 0
23 23.767 29.083 23.0 21.0 479 511 543 575 607 0 13 0 0 0 1 0 0 0 0 0 1
24 -43.03 20.453 20.0 14.0 376 416 456 496 536 1 13 1 0 0 0 0 1 0 0 0 0
25 -35.297 -24.896 10.0 19.0 91 127.8 164.5 201.2 238 1 44 0 0 0 0 0 0 0 0 1 0
26 -54.755 14.368 4.0 14.0 360 396.2 432.5 468.8 505 1 7 0 1 0 0 1 1 0 0 0 0
27 -49.329 33.374 2.0 6.0 379 416.2 453.5 490.8 528 1 35 1 0 0 1 0 1 0 0 0 0
28 57.404 23.822 23.0 16.0 258 300.5 343 385.5 428 1 35 1 0 0 1 0 0 0 0 0 0
29 -22.754 55.408 6.0 9.0 352 391.2 430.5 469.8 509 0 42 0 1 0 0 0 0 1 0 1 0
30 -56.622 73.34 8.0 20.0 288 311 334 357 380 1 58 0 0 0 1 0 0 0 0 0 1
31 -38.562 -3.705 10.0 13.0 159 200.2 241.5 282.8 324 0 39 0 0 0 0 0 1 0 0 0 1
32 -16.779 19.537 7.0 10.0 423 458.2 493.5 528.8 564 1 35 0 1 0 0 0 0 0 1 0 1
33 -11.56 11.615 1.0 16.0 238 275.2 312.5 349.8 387 1 14 0 1 0 0 0 1 0 0 0 0
34 -46.545 97.974 21.0 19.0 339 361.5 384 406.5 429 1 69 0 1 1 0 0 0 0 1 0 1
```

```
35 16.229 9.32 6.0 22.0 397 440.8 484.5 528.2 572 0 12 0 1 0 0 0 0 0 0 1 0
36 1.294 7.349 4.0 14.0 479 509 539 569 599 1 59 0 0 0 0 1 0 0 0 0 1
37 -26.404 29.529 13.0 10.0 315 359.2 403.5 447.8 492 0 17 1 0 0 0 0 0 0 1 0 0
38 4.352 14.685 9.0 11.0 132 171.5 211 250.5 290 0 84 1 0 0 0 0 0 1 1 0 0
39 -50.665 -23.126 22.0 15.0 161 202.2 243.5 284.8 326 1 88 0 1 0 0 0 0 0 0 0 0
40 -22.833 -9.814 22.0 13.0 387 417.2 447.5 477.8 508 1 63 0 0 0 0 0 1 1 0 0 1
41 -71.1 -18.616 18.0 15.0 284 324.5 365 405.5 446 1 76 0 1 0 0 0 1 0 1 0 0
42 -7.849 32.074 10.0 8.0 296 339.8 383.5 427.2 471 1 34 1 0 1 1 0 0 0 0 0 0
43 11.877 -24.933 25.0 22.0 381 406.2 431.5 456.8 482 1 87 0 0 0 1 0 0 0 0 1 0
44 -18.927 -23.73 23.0 24.0 401 431.2 461.5 491.8 522 1 62 1 0 0 0 0 1 0 0 0 0
45 -11.92 11.755 4.0 3.0 432 465 498 531 564 1 85 0 0 0 0 0 1 0 0 0 0
46 29.84 11.633 9.0 25.0 289 329.2 369.5 409.8 450 1 85 0 0 0 0 0 0 0 0 0 0
47 12.268 -55.811 17.0 19.0 451 487.5 524 560.5 597 1 0 0 0 1 0 0 0 1 1 0 0
48 -37.933 -21.613 10.0 21.0 123 167.8 212.5 257.2 302 1 0 1 1 1 0 1 0 0 0 0 0
49 42.883 -2.966 17.0 10.0 98 131.8 165.5 199.2 233 1 76 0 1 0 0 0 1 1 0 0 0
```

Appendix F

Instance MCTOPMTW-100-8

```
1 100 471 2 1 1 5 0 1 2 3 1 3
0 40.0 50.0 0.0 0.0 0 1236
2 45.0 68.0 90.0 10.0 912 925.8 939.5 953.2 967 1 60 0 0 1 0 0 0 0 0 1 0
3 45.0 70.0 90.0 30.0 825 836.2 847.5 858.8 870 1 48 0 0 0 0 0 0 0 0 0 0
4 42.0 66.0 90.0 10.0 65 85.2 105.5 125.8 146 1 29 0 0 0 0 0 0 0 0 0 0
5 42.0 68.0 90.0 10.0 727 740.8 754.5 768.2 782 1 47 0 0 0 0 0 0 0 0 0 0
6 42.0 65.0 90.0 10.0 15 28 41 54 67 1 15 1 1 0 1 0 1 0 0 0 0
7 40.0 69.0 90.0 20.0 621 641.2 661.5 681.8 702 1 53 0 0 0 0 0 0 0 0 0 1
8 40.0 66.0 90.0 20.0 170 183.8 197.5 211.2 225 1 91 0 0 1 0 0 0 0 0 0 0
9 38.0 68.0 90.0 20.0 255 272.2 289.5 306.8 324 1 61 0 0 0 0 0 0 1 0 0 0
10 38.0 70.0 90.0 10.0 534 551.8 569.5 587.2 605 1 19 0 1 0 0 0 0 1 1 1 0
11 35.0 66.0 90.0 10.0 357 370.2 383.5 396.8 410 1 54 0 1 0 0 0 0 0 0 0 0
12 35.0 69.0 90.0 10.0 448 462.2 476.5 490.8 505 1 77 0 0 0 0 0 0 0 0 0 0
13 25.0 85.0 90.0 20.0 652 669.2 686.5 703.8 721 0 77 1 0 0 1 0 0 0 0 0 1
14 22.0 75.0 90.0 30.0 30 45.5 61 76.5 92 1 73 1 0 0 0 0 1 0 0 1 0
15 22.0 85.0 90.0 10.0 567 580.2 593.5 606.8 620 1 62 0 1 0 1 0 0 0 1 0 0
16 20.0 80.0 90.0 40.0 384 395.2 406.5 417.8 429 1 95 0 0 0 0 0 1 0 0 1 0
17 20.0 85.0 90.0 40.0 475 488.2 501.5 514.8 528 1 44 0 0 0 0 0 1 0 0 0 0
18 18.0 75.0 90.0 20.0 99 111.2 123.5 135.8 148 1 84 0 0 0 0 0 0 0 0 0 0
19 15.0 75.0 90.0 20.0 179 197.8 216.5 235.2 254 1 75 0 0 1 0 0 0 0 0 1 1
20 15.0 80.0 90.0 10.0 278 294.8 311.5 328.2 345 1 41 0 0 1 0 0 0 0 1 0 1
21 30.0 50.0 90.0 10.0 10 25.8 41.5 57.2 73 1 20 0 0 1 0 0 0 0 0 0 1
22 30.0 52.0 90.0 20.0 914 926.8 939.5 952.2 965 1 43 0 1 0 1 0 0 0 0 0 1
23 28.0 52.0 90.0 20.0 812 829.8 847.5 865.2 883 1 88 0 0 0 0 0 0 0 0 1 0
24 28.0 55.0 90.0 10.0 732 743.2 754.5 765.8 777 1 24 0 0 1 0 0 0 1 0 1 0
25 25.0 50.0 90.0 10.0 65 84.8 104.5 124.2 144 1 47 0 0 0 0 0 1 1 0 0 1
26 25.0 52.0 90.0 40.0 169 182.8 196.5 210.2 224 1 52 0 0 0 0 0 0 0 0 0 0
27 25.0 55.0 90.0 10.0 622 641.8 661.5 681.2 701 1 60 0 0 0 1 0 0 0 1 0 0
28 23.0 52.0 90.0 10.0 261 274.8 288.5 302.2 316 1 3 0 0 0 0 0 0 0 0 0 0
29 23.0 55.0 90.0 20.0 546 557.8 569.5 581.2 593 1 82 0 0 0 0 0 0 1 1 1 0
30 20.0 50.0 90.0 10.0 358 369.8 381.5 393.2 405 1 92 0 0 1 0 1 1 0 0 0 0
31 20.0 55.0 90.0 10.0 449 462.8 476.5 490.2 504 1 23 1 0 0 0 0 0 0 1 0 0
32 10.0 35.0 90.0 20.0 200 209.2 218.5 227.8 237 1 45 0 1 0 0 1 1 0 0 1 0
33 10.0 40.0 90.0 30.0 31 48.2 65.5 82.8 100 1 45 0 1 0 0 0 0 1 0 1 0
34 8.0 40.0 90.0 40.0 87 104.8 122.5 140.2 158 1 37 0 0 0 0 0 0 0 0 1 1
```

```
35 8.0 45.0 90.0 20.0 751 767.2 783.5 799.8 816 1 87 0 0 0 0 0 0 1 1 0 0
36 5.0 35.0 90.0 10.0 283 298.2 313.5 328.8 344 1 2 0 0 1 1 1 0 0 0 0 1
37 5.0 45.0 90.0 10.0 665 677.8 690.5 703.2 716 1 62 0 0 0 0 0 0 0 0 1 0
38 2.0 40.0 90.0 20.0 383 395.8 408.5 421.2 434 1 25 0 1 0 0 1 0 0 0 0 0
39 0.0 40.0 90.0 30.0 479 489.8 500.5 511.2 522 1 53 1 1 0 0 0 0 0 0 0 0
40 0.0 45.0 90.0 20.0 567 581.2 595.5 609.8 624 1 38 1 0 0 0 0 0 0 0 0 0
41 35.0 30.0 90.0 10.0 264 278.2 292.5 306.8 321 1 35 0 0 0 0 0 0 0 1 0 1
42 35.0 32.0 90.0 10.0 166 183.2 200.5 217.8 235 1 60 0 1 0 0 1 0 0 0 0 0
43 33.0 32.0 90.0 20.0 68 88.2 108.5 128.8 149 1 75 0 0 1 0 0 1 0 0 0 0
44 33.0 35.0 90.0 10.0 16 32 48 64 80 1 55 0 0 0 1 0 0 1 0 1 0
45 32.0 30.0 90.0 10.0 359 372.2 385.5 398.8 412 1 30 1 0 0 1 0 0 1 0 0 0
46 30.0 30.0 90.0 10.0 541 555.8 570.5 585.2 600 1 98 1 1 0 0 0 0 1 0 0 0
47 30.0 32.0 90.0 30.0 448 463.2 478.5 493.8 509 1 91 0 0 1 0 0 0 1 0 1 0
48 30.0 35.0 90.0 10.0 1054 1072.2 1090.5 1108.8 1127 1 74 0 1 0 0 0 0 0 0 0 0
49 28.0 30.0 90.0 10.0 632 647.2 662.5 677.8 693 1 36 0 0 1 1 0 0 0 0 0 0
50 28.0 35.0 90.0 10.0 1001 1017.2 1033.5 1049.8 1066 1 12 0 0 0 1 0 0 0 0 1 0
51 26.0 32.0 90.0 10.0 815 831.2 847.5 863.8 880 1 62 0 0 0 0 0 0 0 1 0 0
52 25.0 30.0 90.0 10.0 725 740.2 755.5 770.8 786 1 19 0 0 0 0 0 0 1 0 0 0
53 25.0 35.0 90.0 10.0 912 926.2 940.5 954.8 969 1 77 0 0 0 1 1 0 0 1 1 0
54 44.0 5.0 90.0 20.0 286 301.2 316.5 331.8 347 1 16 0 0 0 0 0 1 0 1 1 0
55 42.0 10.0 90.0 40.0 186 203.8 221.5 239.2 257 1 46 0 0 0 0 0 0 0 0 1 0
56 42.0 15.0 90.0 10.0 95 110.8 126.5 142.2 158 1 7 1 0 0 1 0 0 0 1 0 0
57 40.0 5.0 90.0 30.0 385 397.8 410.5 423.2 436 1 16 0 1 1 0 0 0 0 0 0 0
58 40.0 15.0 90.0 40.0 35 48 61 74 87 1 8 1 0 1 0 0 0 0 1 0 0
59 38.0 5.0 90.0 30.0 471 486.8 502.5 518.2 534 1 37 0 0 0 0 1 0 1 1 0 0
60 38.0 15.0 90.0 10.0 651 673.2 695.5 717.8 740 1 43 0 1 1 0 0 0 1 0 1 0
61 35.0 5.0 90.0 20.0 562 578.8 595.5 612.2 629 1 47 1 0 0 0 0 0 0 0 0 0
62 50.0 30.0 90.0 10.0 531 550.8 570.5 590.2 610 1 87 1 0 0 0 1 0 1 0 1 0
63 50.0 35.0 90.0 20.0 262 275.8 289.5 303.2 317 1 88 0 0 0 1 0 0 1 0 0 0
64 50.0 40.0 90.0 50.0 171 182.8 194.5 206.2 218 1 5 0 0 0 0 0 0 0 1 0 1
65 48.0 30.0 90.0 10.0 632 647.2 662.5 677.8 693 1 58 1 0 0 0 0 0 0 1 0 0
66 48.0 40.0 90.0 10.0 76 89.2 102.5 115.8 129 1 8 0 0 0 0 0 0 0 0 1 0
67 47.0 35.0 90.0 10.0 826 838.2 850.5 862.8 875 1 17 1 0 0 1 0 0 0 0 0 0
68 47.0 40.0 90.0 10.0 12 28.2 44.5 60.8 77 1 51 1 1 0 0 0 0 1 0 0 0
69 45.0 30.0 90.0 10.0 734 744.8 755.5 766.2 777 1 18 0 1 1 0 1 0 1 1 0 0
70 45.0 35.0 90.0 10.0 916 929.2 942.5 955.8 969 1 58 0 0 0 0 0 0 0 1 0 0
71 95.0 30.0 90.0 30.0 387 404.2 421.5 438.8 456 1 18 0 0 0 0 0 0 0 0 0 0
72 95.0 35.0 90.0 20.0 293 309.8 326.5 343.2 360 1 38 0 0 0 0 1 0 0 1 0 0
73 53.0 30.0 90.0 10.0 450 463.8 477.5 491.2 505 1 72 0 0 0 0 0 0 0 0 0 0
74 92.0 30.0 90.0 10.0 478 496.2 514.5 532.8 551 1 57 0 0 0 0 0 0 0 1 1 0
75 53.0 35.0 90.0 50.0 353 367.8 382.5 397.2 412 1 51 0 0 0 0 0 0 0 0 0 0
76 45.0 65.0 90.0 20.0 997 1014.8 1032.5 1050.2 1068 0 26 0 0 0 1 0 0 0 1 1 0
77 90.0 35.0 90.0 10.0 203 217.2 231.5 245.8 260 1 80 0 1 0 0 0 0 0 0 0 0
78 88.0 30.0 90.0 10.0 574 591.2 608.5 625.8 643 1 97 0 1 0 0 0 0 0 0 0 0
79 88.0 35.0 90.0 20.0 109 124.2 139.5 154.8 170 1 62 1 1 0 0 1 1 0 0 0 0
80 87.0 30.0 90.0 10.0 668 683.8 699.5 715.2 731 1 35 0 0 0 0 0 0 0 0 0 1
81 85.0 25.0 90.0 10.0 769 781.8 794.5 807.2 820 1 20 0 0 0 0 0 1 0 0 0 1
82 85.0 35.0 90.0 30.0 47 66.2 85.5 104.8 124 1 67 0 1 1 0 0 0 0 0 0 0
```

```
83 75.0 55.0 90.0 20.0 369 381.8 394.5 407.2 420 1 73 1 0 1 1 0 1 0 0 0 0
84 72.0 55.0 90.0 10.0 265 283.2 301.5 319.8 338 1 17 0 1 0 1 0 0 0 0 0 1
85 70.0 58.0 90.0 20.0 458 474.2 490.5 506.8 523 1 69 1 0 0 0 0 0 0 1 0 1
86 68.0 60.0 90.0 30.0 555 569.2 583.5 597.8 612 1 5 0 0 0 0 0 0 0 0 0 0
87 66.0 55.0 90.0 10.0 173 189.2 205.5 221.8 238 1 52 0 0 0 0 0 0 0 0 1 0
88 65.0 55.0 90.0 20.0 85 99.8 114.5 129.2 144 1 89 0 0 1 0 1 0 0 0 0 0
89 65.0 60.0 90.0 30.0 645 660.8 676.5 692.2 708 1 43 0 1 1 0 0 0 0 0 0 0
90 63.0 58.0 90.0 10.0 737 753.2 769.5 785.8 802 1 1 1 0 0 0 0 0 1 0 0 0
91 60.0 55.0 90.0 10.0 20 36 52 68 84 1 41 0 0 0 0 0 0 0 0 0 0
92 60.0 60.0 90.0 10.0 836 849.2 862.5 875.8 889 1 23 1 0 0 0 0 0 1 0 1 0
93 67.0 85.0 90.0 20.0 368 386.2 404.5 422.8 441 1 80 0 0 0 0 0 0 1 1 1 0
94 65.0 85.0 90.0 40.0 475 485.8 496.5 507.2 518 1 68 0 0 0 0 0 0 0 0 0 0
95 65.0 82.0 90.0 10.0 285 297.8 310.5 323.2 336 1 14 0 0 1 1 0 0 0 0 0 0
96 62.0 80.0 90.0 30.0 196 206.8 217.5 228.2 239 1 16 0 0 1 1 0 0 0 0 0 0
97 60.0 80.0 90.0 10.0 95 110.2 125.5 140.8 156 1 23 0 0 0 0 0 0 0 1 0 0
98 60.0 85.0 90.0 30.0 561 576.2 591.5 606.8 622 1 57 0 0 0 1 0 1 1 0 0 0
99 58.0 75.0 90.0 20.0 30 43.5 57 70.5 84 1 22 0 0 0 0 0 0 1 0 0 0
100 55.0 80.0 90.0 10.0 743 762.2 781.5 800.8 820 1 5 0 1 1 0 0 0 1 0 0 0
101 55.0 85.0 90.0 20.0 647 666.8 686.5 706.2 726 1 71 0 0 1 0 0 0 1 0 0 0
```