Master Thesis

Automatic Goal Discovery in Subgoal Monte-Carlo Tree Search

Dominik Jeurissen

Master Thesis DKE-21-32

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Data Science and Knowledge Engineering
of the Maastricht University

Thesis Committee:

Prof. Dr. Mark Winands
Dr. Chiara Sironi
Dr. Diego Perez-Liebana

Maastricht University
Faculty of Science and Engineering
Department of Data Science and Knowledge Engineering

August 16, 2021

Preface

This master thesis was written at the Department of Data Science and Knowledge Engineering at Maastricht University in collaboration with the Game AI Group at Queen Mary University in London. This thesis describes my research on detecting subgoals in real-time for usage in Monte-Carlo Tree Search. Parts of this research have been accepted for publication at the 3rd IEEE Conference on Games.

Before I started working on this thesis, I conducted an internship at Queen Mary in a team lead by Dr. Diego Perez Liebana. Diego proposed the topic of this thesis based on some preliminary work by Prof. Simon Lucas. Simon discovered that randomly sampling subgoals for usage in MCTS could be as efficient as using handcrafted subgoals. Intuitively this makes sense, as random subgoals allow MCTS to split the difficult search into more manageable subproblems. But what fascinates me about this approach is that it contains a fundamental idea, any progress is good progress. Therefore, it does not matter how you approach a problem as long as you split it into more manageable problems. Not only is this a good advice in general, but it also helped me tremendously in developing the core idea for this thesis.

I want to thank both Prof. Dr. Mark Winands and Dr. Diego Perez for supervising this thesis and my internship at Queen Mary. Additionally, I want to thank Dr. Chiara Sironi for supervising this thesis. I also like to thank the Game AI Group team for giving me a glimpse into how research is done and for helping me expand my horizon. Finally, I would like to acknowledge my friends and family, who often helped and motivated me.

Dominik Jeurissen
Maastricht, July 04, 2021

Abstract

Monte-Carlo Tree Search (MCTS) is a heuristic search algorithm that can play a wide range of games without requiring any domain-specific knowledge. However, MCTS tends to struggle in very complicated games due to an exponentially increasing branching factor. A promising solution for this problem is to focus the search only on a small fraction of states.

Subgoal Monte-Carlo Tree Search (S-MCTS) achieves this by using a predefined subgoal-predicate that detects promising states called subgoals. However, not only does this make S-MCTS domain-dependent, but defining suitable subgoal predicates is often very difficult due to the complexity of games.

In this thesis, we investigate how S-MCTS can automatically detect subgoals in real-time without requiring any domain knowledge. For this, we review existing methods for discovering subgoals and motivate why quality diversity (QD) algorithms might be better suited for this. Finally, we present an approach for integrating QD-algorithms into S-MCTS, significantly improving its performance in the Physical Travelling Salesman Problem.

Contents

Chapter 1

Introduction

This chapter gives an introduction to this thesis. First, it introduces the concept of artificial intelligence in Section 1.1. Then, Section 1.2 discusses various approaches for designing domain-independent objectives, representing related work to this thesis's main topic. The following Section 1.3, discusses this thesis problem statement and research questions. Finally, Section 1.4 gives an overview of how this thesis is structured.

1.1 Artificial Intelligence in Games

Games are a popular testbed for researching Artificial Intelligence (AI). One challenging problem is finding a good action sequence to control an entity in the game. One of the earliest search technique for this challenge was the $\alpha\beta$ search algorithm (Knuth & Moore, 1975), which was later used in the DEEP BLUE system (Campbell, Hoane, & Hsu, 2002), defeating the world chess champion at the time Garry Kasparov in 1997. Although chess is reasonably complex, the game of Go is far more complex, and algorithms based on $\alpha\beta$ search have not been able to compete with expert human players. However, Monte-Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006; Coulom, 2007) significantly improved the performance in this domain, and it has been later used by ALPHAGO (Silver et al., 2016) to beat one of the strongest professional player Lee Sedol in 2016. MCTS has also been integrated into MUZERO (Schrittwieser et al., 2020) to play a wide range of games, including Go, Chess, Shogi, and a standard suite of Atari games.

1.2 Objectives in General Game Playing

For many games, agents are designed explicitly to play these games using specially designed algorithms and heuristics, which usually cannot be transferred to other games. An example of this is the DEEP BLUE system (Campbell et al., 2002) which can only play chess. General Game Playing (GGP) (Genesereth, Love, & Pell, 2005) instead refers to designing AI that can play multiple games successfully. GGP is considered a necessary stepping stone to create general AI. A general AI is the theoretical concept of an agent that can learn any intellectual task that a human being can.

One major problem in GGP is to design a suitable objective function. Nearly all AI algorithms use an objective function to see if the found solution is a good solution. However, due to the generality of GGP designing one single objective function is essentially impossible. Additionally, poorly designed objective functions can often cause problems when an algorithm finds a way to exploit the objective in unexpected ways. Because of this, a recent trend in AI is to use instead intrinsic objective functions, which do not depend on an external goal. Instead of measuring how close an agent is to solve a specific goal, intrinsic objective functions represent goals that the agent wants to achieve.

One example of this is intrinsic motivation in reinforcement learning (Aubret, Matignon, & Hassas, 2019). In reinforcement learning, an agent continuously acts in an environment and learns to maximise the accumulated future reward. Designing a reward function for reinforcement learning is simple, +1 if the agent wins the game, -1 if it loses, and 0 for any other action. However, such a simple function is nearly useless, as it does not guide the agent. Whenever the agent loses, it receives a reward of -1, and it does not matter how close the agent was to winning. While it is possible to give intermediate rewards, these additional rewards often do not represent the objective, and agents may start focusing more on intermediate rewards than winning.

One form of intrinsic motivation is curiosity, in which the agent gets rewarded when it visits states that it has not seen before. This reward function has two advantages. First, it produces a dense reward space, meaning any action the agent executes returns some form of meaningful reward. Secondly, it encourages the agent to explore the entire game. Although this reward function does not explicitly mention the goal of winning the game, in most games exploring the game will automatically lead to a win. This type of intrinsic motivation has been successfully used in Go-Explore (Ecoffet, Huizinga, Lehman, Stanley, & Clune, 2019) to solve Montezuma's revenge, a game in which traditional objective functions failed.

Another approach for intrinsic objective functions is novelty. Novelty is measured by comparing new solutions to old solutions. A new solution gets a higher novelty score the more it differs from the old solutions. In one instance, authors adapted Monte-Carlo Tree Search to use novelty for the selection step (Baier & Kaisers, 2021). Although the resulting agent's win rate was marginally lower than a traditional MCTS agent, the novelty enhanced agent required much fewer simulation steps to achieve good results.

Finally, the iterated width algorithm (Geffner & Geffner, 2021) is a breadth-first search that uses novelty to prune many actions. Whenever the algorithm visits a new state, it converts the state into a set of boolean features. Any newly generated state is ignored if it does not change one of the boolean features compared to the previous state. Although this is a relatively simple algorithm, it showed excellent results in the GVG-AI competition. The GVG-AI competition evaluates agents on a wide range of games to measure the agent's capabilities in GGP. Although the used boolean features have to be manually defined, these results show that novelty can be much more effective than using traditional objective functions.

1.3 Problem Statement and Research Questions

Monte-Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006; Coulom, 2007) is a heuristic search algorithm that achieves good performance in a wide range of games without requiring any domain knowledge (Browne et al., 2012). Compared to algorithms like minimax, MCTS can often handle more complex games with a high branching factor. This can be attributed to MCTS investing the available computation-time to more promising paths instead of equally distributing it. However, as the complexity of a game grows, MCTS has to sample more trajectories to estimate an action's value. This leads to the issue that MCTS essentially plays randomly in very complicated games.

Subgoal-MCTS (S-MCTS) (Gabor, Peter, Phan, Meyer, & Linnhoff-Popien, 2019) tries to solve this issue by introducing a low-level search for finding subgoal-states. Given a subgoal predicate, any state that fulfils the predicate is treated as a subgoal. S-MCTS uses a low-level search for growing the search tree, and a high-level MCTS search then only chooses between detected subgoals. The core idea is that subgoals allow the search to optimize partial trajectories instead of the whole trajectory.

One problem with S-MCTS is that the subgoal predicate has to be provided by the developer, which makes the algorithm domain-dependent. Additionally, it is often quite challenging to provide a good subgoal predicate. For example, if the trajectories between subgoals are short, the technique will have the same problems as MCTS. Additionally, if some parts of the search space have no subgoals defined, S-MCTS will not consider this subspace at all. Although this might be a desirable property in some cases, it can also cause S-MCTS to miss important parts of the search space. As such, the problem statement of this master thesis is:

How can we automatically detect subgoals in real-time for usage in Subgoal-MCTS?

To address this problem, it is promising to first investigate what a useful subgoal for S-MCTS is and how these subgoals can be detected. This gives us the following two research questions:

1. *What states qualify as useful subgoals for Subgoal-MCTS?*
2. *What metrics can be used to detect the subgoals?*

With a suitable definition for subgoals, the next step is to develop an algorithm that can find these states.

3. *How can a search algorithm quickly detect subgoals?*

Once an algorithm for detecting subgoals is found, the last step is to integrate the subgoal-detection into Subgoal-MCTS.

4. *How can subgoal-detection be integrated into Subgoal-MCTS to improve its performance?*

1.4 Thesis Overview

The remainder of this thesis is structured as follows. First, Chapter 2 describes various search algorithms that are referenced throughout this thesis. Next, in Chapter 3 we define what a good subgoal is and motivate why it may not be necessary to search for them explicitly. Based on the previous discussion, we show how subgoal MCTS can be adapted to discover subgoals quickly. Section 3.6 introduces the second adaption of subgoal MCTS by tweaking a small but essential detail. The newly introduced algorithms are evaluated in Chapter 4, which describes the setup for all experiments conducted and the results. Finally, Chapter 5 links the results of this thesis to the research questions and the research problem. Finally, the chapter concludes this thesis by providing ideas for future research.

Chapter 2

Methods

This chapter provides an introduction to various search techniques that are important for understanding this thesis. First, Section 2.1 introduces the MCTS algorithm. Then, the algorithm, which this thesis is focussing on, called S-MCTS, is introduced in Section 2.2. Next, genetic algorithms are discussed in Section 2.3. Finally, Quality Diversity algorithms are discussed in Section 2.4.

2.1 Monte-Carlo Tree Search (MCTS)

Monte-Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006) (Coulom, 2007) is a anytime search-algorithm that uses sampling to approximate the value of actions. It gradually builds up a search-tree to track previous results and focus on more promising parts of the search space. Initially, the search-tree contains only the root-node representing the initial state of the search. To advance the search, MCTS executes four steps (Chaslot, Winands, Herik, Uiterwijk, & Bouzy, 2008), which are shown in Figure 2.1.

Selection
In the selection step, MCTS traverses the tree starting from the root until it reaches a node that has not not been fully expanded yet. A node is fully expanded when there is one child for every action available in this node. The traversal is guided by a selection policy that should balance between exploitation and exploration. A commonly used selection policy is the Upper Confidence Bound adapted for trees (UCT) (Kocsis & Szepesvári, 2006).

Play-out
In the play-out step, the partial trajectory is completed by (semi)-randomly sampling actions. This step aims to reduce the bias that arises if the search is only looking at unfinished trajectories.

Expansion
The expansion step adds one or more actions to the previously selected node.

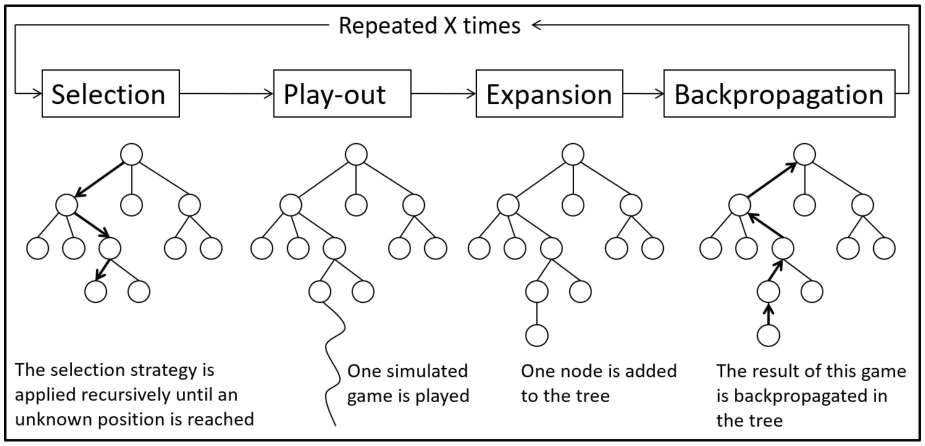

Repeated X times
Selection
Play-out
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

Figure 2.1: The four steps of an Monte-Carlo Tree Search adjusted from (Chaslot et al., 2008).

The added nodes can then be used in further iterations for tracking results to guide the search.

Backpropagation
After completing the trajectory, MCTS has access to the accumulated rewards from the environment. These rewards are then backpropagated through all visited nodes to update the expected value of the actions.

2.2 Subgoal-MCTS

In many environments, the agent has to decide between numerous actions, and it becomes infeasible to explore every individual action thoroughly. In addition, many actions may even achieve similar results, and as such, finding the most optimal trajectory may not be necessary. Subgoal-MCTS (S-MCTS) (Gabor et al., 2019) adapts MCTS to better work in these kinds of environments. It splits the problem into two sub-problems, a low-level search for finding short action sequences called macro-actions and a high-level MCTS-search that chooses between these macro-actions.

It is noteworthy that macro-actions have been incorporated into MCTS in various ways. The goal of macro-actions in all cases is to simplify the original problem somehow. One such example is to repeat a selected action n times, where n is a predefined parameter. By repeating an action multiple times, MCTS can search much deeper than what would usually be possible. The tradeoff is that it reduces the flexibility of the search. S-MCTS, however, focuses on searching for macro-actions that lead to predefined subgoals. That means S-MCTS can be more flexible, but the tradeoff is that the low-level search first has to find the macro actions. The idea is that by searching for trajectories between subgoals, the search is split into multiple smaller problems. Instead of running one MCTS search to optimize the whole trajectory, the low-level search greedily searches for partial trajectories, which are then optimized globally by the high-level search.

2.2.1 High-Level Search

The high-level search in S-MCTS is a traditional MCTS search, with some slight tweaks to incorporate the macro-actions and the low-level search. Since an edge in the search tree now represents a macro-action, each edge requires executing multiple actions when traversing the tree. Additionally, the search has to keep track of the accumulated reward while executing a macro-action.

The most significant change is that the expansion step of MCTS is replaced with a low-level search for finding suitable macro-actions. For every leaf node in the search tree, the data for a local low-level search is stored. The goal of the low-level search is to find macro-actions that lead to subgoals. Whenever a leaf node is visited, the low-level search is advanced by one step. Note that for each step, the low-level search only has to find one subgoal at a time. Once the low-level search cannot find any new subgoals anymore, the leaf node is considered fully expanded, and all found subgoals are added to the leaf node. Algorithm 1 contains the pseudocode for the discussed high-level search.

Algorithm 1 Subgoal-MCTS

```
Require:
    Root of the search tree $r$
    Initial state $s_0$
 1: procedure S-MCTS
 2:     while computation budget remaining do
 3:         $(node, state) \leftarrow Selection(r, s_0)$
 4:         $(done, state) \leftarrow LowLevelSearch(node, state)$
 5:         if $done$ then
 6:             $S \leftarrow getDiscoveredSubgoals(node)$
 7:             $AddSubgoalNodes(node, S)$
 8:         end if
 9:         $(state, rewardSum) \leftarrow Play\text{-}Out(state)$
10:         $Backpropagate(node, rewardSum)$
11:     end while
12: end procedure
```

2.2.2 Low-Level Search

The low-level search uses random sampling to find trajectories that lead to predefined subgoals. For this, the low-level search has access to a predefined subgoal predicate, which detects whether a given state is a subgoal. Whenever the low-level search is advanced, the search will uniformly sample actions until a subgoal is found or a horizon is reached. States found when reaching the horizon are still treated as subgoals.

The search tracks all discovered subgoals and the corresponding best macro action. Whenever a new subgoal has been found, the corresponding state is returned for usage in the high-level search. If a previously discovered subgoal was found, the sampling process will continue until a new subgoal has been found or when a threshold is reached. Once the threshold is reached, the low-level search is considered complete, and all found subgoals will be added to the high-level search tree. Algorithm 2 shows the pseudocode for the low-level search.

Algorithm 2 Low-Level Search with Random Sampling

```
Require:
    Subgoal predicate $p$
    Search horizon $H$
    Search threshold $T$
 1: procedure LowLevelSearch(parentNode $n$, currentState $s_t$)
 2:     for $i \leftarrow 1$ to $T$ do
 3:         $s_i \leftarrow s_t$
 4:         $m_t \leftarrow \{\}$
 5:         do                                              ▷ Sample macro action
 6:             $a \leftarrow sampleAction(s_i)$
 7:             $s_i \leftarrow forwardModel(s_i, a)$
 8:             $m_t \leftarrow m_t \cup \{a\}$
 9:         while $|m_t| < H \wedge \sim p(s_i)$
10:         $S \leftarrow getDiscoveredSubgoals(n)$
11:         if $\exists m' \in S : forwardModel(s_t, m') = s_i$ then ▷ Duplicate Subgoal?
12:             if $reward(s_t, m_t) > reward(s_t, m')$ then
13:                 $S \leftarrow (S \setminus m') \cup \{m_t\}$
14:             end if
15:         else                                            ▷ We found a new subgoal
16:             $S \leftarrow S \cup \{m_t\}$
17:             return $(False, s_i)$
18:         end if
19:     end for
20:     return $(True, s_t)$
21: end procedure
```

2.3 Genetic Algorithm

Genetic Algorithms (GA) (Dasgupta & Michalewicz, 1997) are black-box optimizers that are inspired by the process of natural selection. In all Genetic Algorithms, a population of candidate solutions called genomes are iteratively evolved toward better solutions. A population usually starts with a set of randomly initialized genomes, and in each iteration called generation, the population is replaced with a newly generated one. Note that genomes from the old population may still be contained in the new population. In order to generate a new population, genetic algorithms use biologically inspired operations such as mutation and crossover.

In each generation, the fitness of the genomes is evaluated using a fitness function. The fitness function uses the information encoded in the genome to assess how good the solution is that the genome represents. The more fit genomes are stochastically selected from the current population to reproduce and form the new population. The reproduction is done by recombining two existing genomes using a crossover operator, and offspring may be adapted using the mutation operator.

A common genome representation for games is a sequence of actions with a fixed size (Gaina, Devlin, Lucas, & Perez-Liebana, 2020). The main reason for this representation is that it is easy to evaluate. The fixed size also allows for simple crossover and mutation operations. One crossover operation is the one-point crossover, in which a random point is chosen that splits the sequence in two. The two parents then exchange one part of their genome sequence for creating two new offspring. A simple mutation operation is to randomly change elements of the sequence with a probability of $1/n$, where n is the length of the sequence. Thus, this mutation operation will, on average, change one action.

2.4 Quality Diversity

Many optimization algorithms focus on finding a policy/behaviour that maximizes performance. Instead, Quality Diversity (QD) (Pugh, Soros, & Stanley, 2016) algorithms try to find as many diverse behaviours as possible while also seeking the behavioural niche's best-performing individual.

QD algorithms make use of a concept called behaviour characterization (BC). The BC is usually a vector representing the sequence of actions taken by an individual, but it can also include additional information about the individual. For example, if the goal is to navigate a maze, the behaviour-vector could simply be the agent's final position. Another vital part of QD algorithms is an archive of previously discovered behaviours. With the archive, the novelty of a new behaviour can be computed by comparing the distance to previous behaviours; the larger the distance, the more novel the behaviour is.

QD algorithms seek to explore the entire behaviour space, which can help avoid deceptive fitness functions used in traditional optimization algorithms. Although this can be achieved using novelty alone (Lehman & Stanley, 2008), QD algorithms also enforce a fitness constraint to find novel and high performing solutions.

2.4.1 Novelty Search

Novelty Search (NS) (Lehman & Stanley, 2011a) is a QD algorithm that works by replacing the fitness reward in evolutionary algorithms with a novelty reward. The novelty of a genome is computed by first computing its behaviour-vector, which depends on the problem domain. Then, with the given behaviour vector, novelty is computed by taking the sum of distances to the k-nearest behaviours in the archive, which are closest to the genomes behaviour.

After each generation, some of the newly generated genomes are added to the archive. There exist various strategies on how to do this. One approach uses a novelty threshold and adds any individual to the archive that exceeds this threshold. Another approach is to add any genome to the archive with a given probability. Often, the archive's size is also constrained to save computation time; this can be done, for instance, by replacing the behaviour with the lowest novelty.

2.4.2 Novelty Search with Local Competition

Since Novelty Search only rewards novelty, the resulting solutions will be diverse but often fail to solve the user's objective. Novelty Search with local competition (NS-LC) (Lehman & Stanley, 2011b) tries to solve this problem using a multi-objective evolutionary algorithm. The two metrics used in NS-LC are novelty and fitness. Thus, NS-LC tries to find solutions that are diverse and have high performance.

The novelty metric is computed as in Novelty Search by computing the sum of distances to the k-nearest neighbours in the archive. The fitness metric is defined as the number of k-nearest behaviours with a lower reward than the genome. The addition of the fitness metric is relatively cheap since the nearest neighbours are already needed for computing the novelty metric.

Chapter 3

Subgoal Discovery in Subgoal-MCTS

In this chapter, we present our approach for discovering subgoals in S-MCTS. To do this, we first discuss in Section 3.1 what an optimal subgoal is. In Section 3.2, we review existing approaches for discovering subgoals and motivate why quality diversity algorithms might be better suited for this. Section 3.3 and Section 3.4 discuss the entire algorithm. Lastly, Section 3.5 describes how an agent can use S-MCTS for real-time applications.

3.1 Optimal Subgoals

The optimal subgoal is a state that lies on an optimal trajectory, meaning there is no trajectory with a higher return. The goal of subgoal discovery is to find subgoals that are optimal or close to optimal. It does not matter if all subgoals are optimal, as long as at least one optimal subgoal is found, given that we start searching from an optimal subgoal. Figure 3.1 shows how this property enables S-MCTS to find optimal trajectories.

3.2 Any Subgoal is Sufficient

Existing approaches for subgoal discovery implicitly search for optimal subgoals. For example, in one approach, bottlenecks are used as subgoals (McGovern & Barto, 2001). Bottlenecks are states which are frequently visited on highly rewarding trajectories, such as a door in a gridworld. Since bottlenecks have to be visited to reach the goal, they are automatically an optimal subgoal. Another approach is to find states with high empowerment (Gregor, Rezende, & Wierstra, 2016). Empowerment measures how much influence actions have on the future states that the agent can reach. High empowerment ensures that

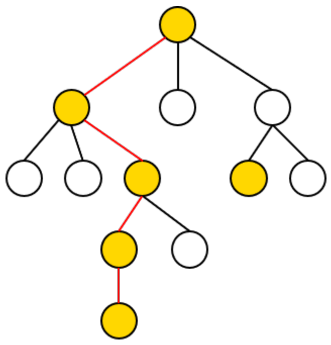


Figure 3.1: Example of how optimal subgoals (Marked golden) can enable S-MCTS to find optimal trajectories (Marked red).

the agent has many options available, making it more likely to find an optimal trajectory.

Although these approaches are viable techniques for discovering subgoals, they either require much sampling or a learning algorithm. Since the goal is to make S-MCTS domain-independent, we cannot afford to spend much time on detecting a single subgoal. Additionally, finding optimal subgoals based on one metric is very difficult due to a sparse distribution. For example, bottlenecks like doors are often quite spread out throughout the environment. However, it may not be necessary to search for optimal subgoals explicitly. Subgoals in S-MCTS are effectively a way to prune many trajectories. For example, given a room with two doors, we may define one subgoal for each door to sufficiently explore all rooms. By doing this, we prune away all trajectories except two. However, if only one trajectory is used, the high-level search loses the ability to explore the entire state-space. What that means is that we only have to find a set of trajectories that enable the high-level search to sufficiently explore the state-space. It does not even matter if the trajectory leads immediately to the door, as long as we eventually move through it. Figure 3.2 visualizes this idea.

The question that remains is how we can select subgoals in order to explore the state-space. Fortunately, quality diversity algorithms already solve this problem. Although QD algorithms focus on finding diverse behaviours, we can always define the behaviour as the state reached by a trajectory. We can then use any QD algorithm to find a set of trajectories that lead to novel states with high fitness. The resulting trajectories can then be used as subgoals in the S-MCTS search.

3.3 Quality Diversity MCTS

In this section, we describe our approach for integrating QD algorithms into S-MCTS, we call the resulting family of algorithms Quality Diversity MCTS (QD-MCTS).

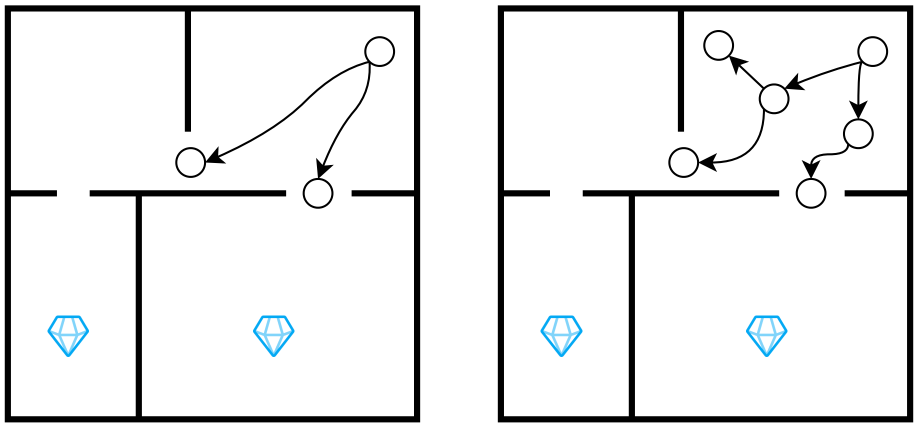


Figure 3.2: An example of how searching for optimal subgoals may not be required. The left map shows how searching for bottlenecks would result in subgoals that find doors. The map on the right shows how intermediate subgoals can fulfil the same goal as long as the subgoals cover enough space.

In general, the S-MCTS algorithm stays the same, except that we replace the low-level predicate based search with a QD-search. Furthermore, we assume that S-MCTS can access a behaviour-characterization function that maps a trajectory onto a latent space representing the behaviour.

For every leaf node in the high-level search tree, we store data for a low-level QD-search. The goal of the QD-search is to find a set of trajectories that lead to high rewards and novel states. Note that the novelty is measured locally, meaning subgoals from the high-level tree are not used in the low-level QD-search.

Since QD-MCTS requires no explicitly defined subgoals as in S-MCTS, there are a few issues that need to be addressed. In comparison to the low-level search in S-MCTS, the QD-search does not return one subgoal at a time. Instead, we assume that the QD-search is an anytime algorithm, and whenever we visit a leaf node, the QD-search is advanced by one iteration. Once the low-level search has been advanced sufficiently often, the search is considered finished, and the best trajectories will be added as subgoals to the leaf node. At last, in S-MCTS the low-level search always returns a found subgoal for usage in the simulation and backpropagation step of the high-level search. Since we no longer return subgoals until the low-level search is done, we instead assume that the low-level search samples precisely one trajectory per step. The state reached by the sampled trajectory is then used in the high-level search. Note that this may cause problems with the value estimation in the high-level search since most of the trajectories sampled by the QD-Search will not be used as subgoals.

3.4 Low-Level Search

Because we use MCTS for the high-level search, we decided to also use it for the low-level search. To ensure that the low-level MCTS search can find novel states, we adapted the reward function to better represent this goal. Concretely we use as a reward the latent-distance to the subgoal node that contains the low-level search. Given a distance-metric $d(s_1, s_2) \rightarrow \mathbb{R}_0^+$, and the behaviour vector of the subgoal node s_g we define the reward at each timestep as follows:

$$r(s_i, s_{i+1}) = d(s_{i+1}, s_g) - d(s_i, s_g) \tag{3.1}$$

The reward function is positive when the distance to the high-level subgoal increases and becomes negative if we get closer. Note that the function does not consider how long it takes to reach a certain distance, resulting in idling where the agent moves back and forth between two positions. To fix the idling, we use a discounted sum of rewards with a discount factor below 1. The discount factor reduces rewards that are further in the future, meaning if an agent idles, it will get fewer rewards than if it moves quickly. This procedure results in an MCTS algorithm that focuses on finding trajectories that quickly move away from the high-level subgoal.

As described in Section 3.3, once the low-level search has been advanced a set amount of time, it has to return a set of subgoals. Algorithm 3 contains the pseudocode for the subgoal selection. It works by first selecting a set of subgoal candidates, which are all trajectories with a specific length. The length is kept the same to make the comparison easy. Subgoals are then iteratively added to an archive by evaluating candidates using a relative novelty and reward score. We use the same scores as used in Novelty Search with local competition (Lehman & Stanley, 2011b). The novelty score is the sum of distances to the k nearest subgoals, and the reward score is the number of k nearest subgoals with a lower reward than the candidate. Both scores are normalized between 0 and 1 and then combined to a final score using a weighted sum with $\alpha = 0.5$. We add subgoals until we selected a percentage of all available candidates. For this thesis, we select 2% of the candidates.

3.5 Real-Time Application

For real-time applications, an agent usually has 40 milliseconds (Perez-Liebana et al., 2016; Pepels & Winands, 2012) to respond to the last observation. This time window is often not sufficient to finish more than the subgoal-search in the root. The result is that QD-MCTS cannot search deep enough to benefit from the pruned trajectories. This problem can be solved by reusing the search tree of the high-level search. Tree reuse is a common technique in MCTS (Soemers, Sironi, Schuster, & Winands, 2016) and can be easily adapted for QD-MCTS.

Whenever the agent has to act, it first searches in the root for the best macro-action to execute. For example, this can be the macro-action with the highest mean reward or the highest number of explorations. The agent will then execute

Algorithm 3 Subgoal selection

```
Require:
    Root of the low-level MCTS search tree $r$
    Length of a macro action $L$
    Number of neighbours $k$ for novelty and reward score
    Percentage $p$ of trajectories to select
    Weighting for rewards $\alpha$
 1: procedure SELECTSUBGOALS
 2:     $cand \leftarrow SelectNodes(r, L)$                      ▷ Subgoal candidates
 3:     $c_{max} \leftarrow c \in cand$ with highest reward
 4:     $subgoals \leftarrow \{c_{max}\}$                        ▷ Start with best candidate
 5:     $sCount \leftarrow p \cdot |cand|$                       ▷ Subgoal Count
 6:     while $|subgoals| < sCount$ do
 7:         for $c_i$ in $cand$ do                               ▷ Update scores
 8:             $neighbours \leftarrow kClosest(subgoals, c_i, k)$
 9:             $rScore \leftarrow rewardScore(neighbours, c_i)$
10:             $nScore \leftarrow noveltyScore(neighbours, c_i)$
11:             $assignScore(c_i, \alpha \cdot rScore + (1 - \alpha) \cdot nScore)$
12:         end for
13:         $c_{best} \leftarrow c \in cand$ with highest score
14:         $subgoals \leftarrow subgoals \cup \{c_{best}\}$
15:     end while
16:     return $subgoals$
17: end procedure
```

all low-level actions contained in the macro-action one by one. One approach for tree reuse replaces the root with the child corresponding to the chosen macro-action; The subtree below this child is retained and all its statistics, while the remaining branches are discarded. After executing the macro-action, the tree can be used for further searches.

The problem with executing the whole macro-action is that the agent wastes a lot of computation time since executing one low-level action corresponds to 40ms of computation time. Therefore, it would be better if the agent can continue searching while executing the macro action. This can be done by continuously removing the last executed low-level action from the corresponding macro-action. Once the macro-action is empty, the root is replaced as previously described. Thus, by removing one low-level action at a time, the search tree is advanced by exactly one timestep, making it possible to continue searching without wasting computation time. Algorithm 4 contains the pseudocode for a real-time QD-MCTS agent. Note that the approach described in this section applies to any algorithm described in this thesis that is based on S-MCTS.

Algorithm 4 Real-Time QD-MCTS Agent

```
Require:
    Root of the search tree $r$
    Last observed state $s$
 1: procedure QD-MCTS AGENT
 2:     Run QD-MCTS with $r$ and $s$
 3:     if $ChildCount(r) > 1$ then
 4:         $node \leftarrow SelectBestChild(r)$
 5:         $ReplaceChildren(r, \{node\})$
 6:     end if
 7:     $node \leftarrow FirstChild(r)$
 8:     $macroAction \leftarrow GetMacroAction(node)$
 9:     $nextAction \leftarrow RemoveFirstElement(macroAction)$
10:     if $|macroAction| = 0$ then
11:         $r \leftarrow node$
12:     end if
13:     return $nextAction$
14: end procedure
```

3.6 Adapting Subgoal MCTS

In this section, we propose a second alternative to the S-MCTS algorithm described in Section 2.2. In S-MCTS, the expansion step is replaced with a low-level search that uniformly samples actions until it found a subgoal or until a horizon is reached. However, when the horizon is reached, the found state will still be treated as a subgoal. The result is a search that treats nearly every state as a subgoal.

The main advantage of S-MCTS is that subgoals enable the algorithm to prune many trajectories. However, by treating nearly every state as a subgoal, we essentially lose this vital property. It is noteworthy that the authors of the original S-MCTS paper have shown that S-MCTS surpasses MCTS in a grid-world and the game Tetris. However, both games have small branching factors, and randomness does not affect the reachable states. Thus, when limiting the horizon, the random search can only find a handful of unique states, for which it only keeps the best trajectory. As such, the reason for S-MCTS success in these games is not the subgoal-predicate. Instead, the success results from the pathfinding algorithm that is implicitly implemented by the random search.

Since our goal is to compare QD-MCTS with S-MCTS, we propose to change S-MCTS to ignore any trajectory that reaches the horizon. Note that this will cause S-MCTS to act randomly if it cannot find subgoals in the given horizon. However, we think that this small change will already improve the results of S-MCTS significantly for complex games.

Chapter 4

Experiments

In this chapter, we evaluate the performance of the previously proposed algorithms. For this, we first introduce the PTSP competition in Section 4.1. Then, Section 4.2 explains the experiment setup and how all involved algorithms were optimized.

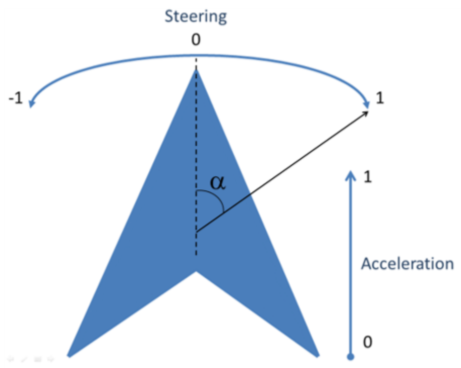
The first experiment is described in Section 4.3, and it evaluates how the agents behave with different number of waypoints. The second experiment described in Section 4.4 uses a stochastic environment to test the robustness of QD-MCTS. Finally, Section 4.5 analyzes how QD-MCTS finds solutions compared to a vanilla MCTS algorithm.

4.1 Physical Travelling Salesman Competition

The physical travelling salesman problem (PTSP) (Perez Liebana, Rohlfshagen, & Lucas, 2012) is an adaptation of the well-known travelling salesman problem (TSP). In the TSP, a salesman has to visit a series of cities, and the goal is to minimize the travelling time. More generally, starting from a node in a given graph with costs assigned to each edge, the goal is to find the path with the lowest cost that visits each node at least once. In the TSP, moving between nodes is represented by the cost of the edge. Therefore, the PTSP adapts TSP by removing this simplification. Instead of working with a simplified graph, an agent has to steer a ship to reach a set of waypoints scattered around a map. As a result, the PTSP is much harder to solve since the agent has to find a good ordering of waypoints and steer the ship correctly to minimize the travelling time.

To evaluate the performance of our proposed algorithms, we used the framework of a competition based on the PTSP[1]. In the competition, an agent has 1000 steps to reach a waypoint. After reaching a waypoint, the time limit is reset back to 1000. To move, the agent can control the ship's thruster and

[1]The source code, as well as all experiments, can be found at: https://github.com/CommanderCero/AutoSubgoalMCTS_PTSP



Action ID	Acceleration	Steering
0	No (0)	No (0)
1	No (0)	Left (-1)
2	No (0)	Right (1)
3	Yes (1)	No (0)
4	Yes (1)	Left (-1)
5	Yes (1)	Right (1)

Figure 4.1: The forces applied to the ship and the corresponding actions in the PTSP competition. Adapted from (Perez Liebana et al., 2012).

rotate the ship left and right. Figure 4.1 shows the available actions in more detail. Although the agent only has to decide between six actions, it takes many steps to move between waypoints. In fact, the search depth required for this environment is so high that we had to lower it by repeating a chosen action fifteen times before moving to the next one. For this thesis, we use a set of ten training maps that were provided by the competition to test algorithms before submitting them. All maps, by default, contain ten waypoints that are scattered around the map. Note that the placement is not randomized. Figure 4.2 shows a map rendered by the framework.

Many strong solutions have already been proposed for this environment. For example, one approach which we use for comparison employs MCTS and can reliably reach all waypoints (Perez et al., 2014). However, although MCTS is used to steer the ship, the authors precomputed the order of waypoints and integrated them into the rewards. By integrating the waypoints into the rewards, the authors created a dense reward landscape enabling MCTS to focus on finding the most efficient trajectory between waypoints. In addition to precomputing the waypoint order, the proposed solution also repeats chosen actions several times to lower the search depth.

4.2 Experiment Setup and Optimization

Chapter 3 and Section 3.6 proposed two alternatives to the original S-MCTS algorithm from Section 2.2. We evaluate the alternatives in the PTSP framework without providing them with any domain knowledge. By providing no domain knowledge, we can test if the algorithms can still find waypoints despite pruning many trajectories. If that is the case, then the added search-depth should enable the subgoal-based algorithms to beat traditional algorithms. As such, the reward function used for all upcoming experiments returns 10 when a new waypoint is reached and otherwise -1.

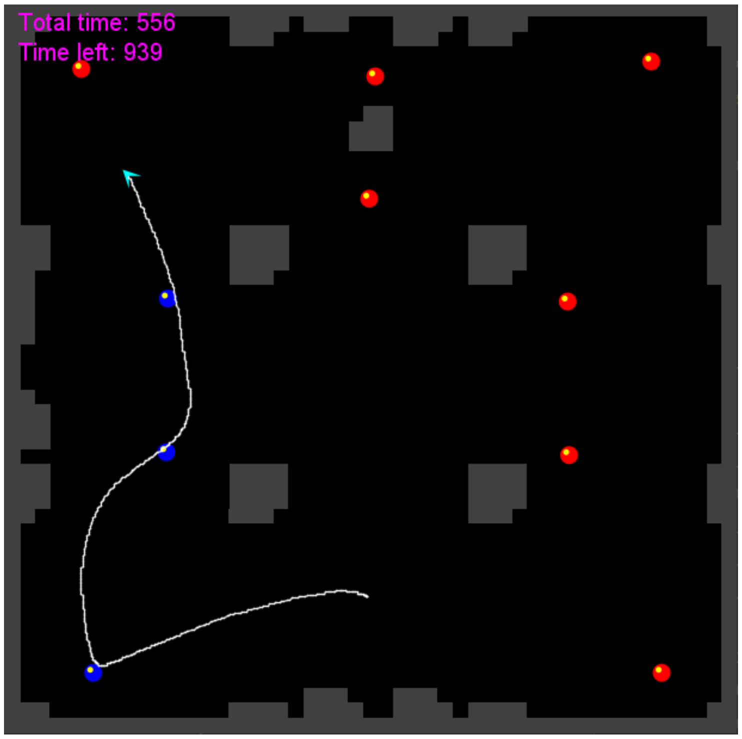


Figure 4.2: A map from the PTSP competition. The ship is represented by the triangle, while the white line shows the path taken by the ship. Red circles represent waypoints, and they turn blue once the ship reaches them.

The algorithms that will be compared are the vanilla MCTS algorithm, a vanilla genetic algorithm as described in Section 2.3, S-MCTS, the modified S-MCTS from Section 3.6 called MS-MCTS, and the novelty-based S-MCTS algorithm from Chapter 3 called QD-MCTS. Note that all algorithms reuse the search results. S-MCTS and MS-MCTS are given access to a subgoal predicate that evenly distributes a grid of subgoal points over the map. The predicate detects a subgoal whenever a ship is in a certain radius to a subgoal point. The velocity of the ship is ignored when detecting a subgoal. The behaviour vector used for QD-MCTS contains only the ship's position.

The rule-based MCTS algorithm (Perez et al., 2014) described in Section 4.1 is also used for comparison. Note that it is unlikely that any algorithm can beat the rule-based algorithm since it is nearly impossible to find the optimal ordering of waypoints without access to domain knowledge.

4.2.1 Optimization

To provide a fair comparison, we optimized the hyperparameters of all algorithms, excluding the rule-based version since it only serves as an estimate for the upper limit. For the optimization, we have used a bayesian optimization with Gaussian processes (Snoek, Larochelle, & Adams, 2012). Appendix B contains all the parameters that have been optimized and the corresponding bounds. For each algorithm, the optimizer tested 50 parameter combinations. A parameter choice is evaluated by running the corresponding algorithm ten times for each of the ten maps available in the PTSP framework. For each run i, a score is computed based on the number of waypoints reached w_i and the total number of steps s_i the algorithm has taken:

$$score(i) = w_i + \frac{s_i}{1000} \tag{4.1}$$

Subsequently, the mean score of all runs is used as the score for the corresponding parameter choice. Since an agent has exactly 1000 steps to reach a waypoint, the score will be -1 if no waypoint is reached. However, whenever a waypoint is reached, the timer resets back to 1000. Thus the theoretical optimum is to reach each waypoint in 0 steps and get a score equal to the number of waypoints, which is 10 in this case.

4.2.2 Results

Figure 4.3 shows how the optimized algorithms performed on each map. Although vanilla MCTS and QD-MCTS seem to have nearly identical scores, when looking at each map individually, we can see that QD-MCTS outperforms vanilla MCTS on map 45 and 56. Map 56 is a fully-fledged maze and is the most challenging map, in which QD-MCTS managed to visit one waypoint more on average. Map 45 contains a cave system with multiple small open areas connected by narrow corridors, and each area contains one waypoint. QD-MCTS outperformed all algorithms on this map by a large margin, probably because the increased search depth made it easier for QD-MCTS to find the areas that still contain a waypoint. Apart from these two maps, QD-MCTS and vanilla MCTS behave nearly identical. One possible explanation for this is that the remaining maps contain large open areas, making it easy to find waypoints using only sampling. This explanation is supported by the fact that all optimized algorithms rely heavily on sampling, which we will discuss later in more detail.

Another interesting observation is that S-MCTS and MS-MCTS behave identically in terms of waypoints reached. However, MS-MCTS requires more steps on average to reach a waypoint. One weakness that we have noticed is that MS-MCTS focuses too much on reaching a subgoal. The result is that the algorithm takes any manoeuvre to reach a subgoal, even if it reduces the ships speed to zero. We may solve this by also considering velocity in the subgoal-predicate. However, this shows that predefined subgoals require extensive engineering to avoid these kinds of problems. S-MCTS is not affected by this problem because

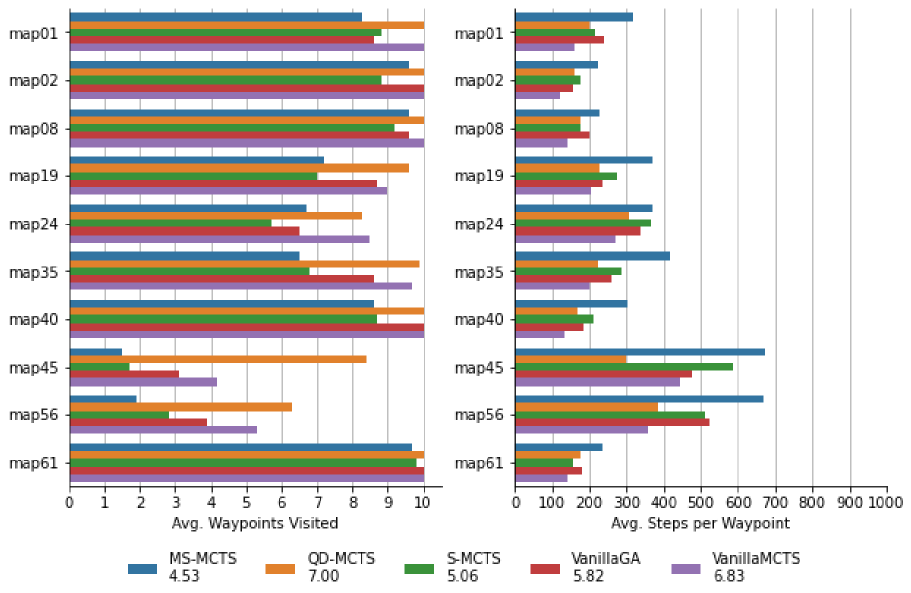


Figure 4.3: The results that the optimized algorithms achieved, the optimization score is included in the legend. **(Left)** The average number of waypoints an algorithm found on each map. **(Right)** The average number of steps an algorithm needed to reach a waypoint.

the low-level search treats any state as a subgoal. While this increases the branching factor significantly, it helps negate any problems with the predefined subgoal predicate.

At last, we have to address that the parameters found for all algorithms are pretty unexpected. For example, the exploration rate for QD-MCTS for the low and high-level search is around 15. In contrast, one popular default choice for the exploration rate is $\sqrt{2}$. Similar observations can be made for all the other algorithms, as can be seen in Appendix B. However, Figure 4.4 shows how the different parameter choices affected the final score of QD-MCTS. As it turns out, the only parameter that seems to have a significant impact is the rollout depth. Since a high rollout depth seems to correspond with a high score, it means that sampling plays a vital role in the PTSP environment. Similar observations can be made for the other algorithms, as can be seen in Appendix B.

From this analysis, we can conclude that the odd parameter choices are probably by pure chance. It is at least correct to choose lower exploration rates and get a similar score, as Figure 4.4 shows. However, the fact that sampling is so efficient in PTSP is counterproductive for these experiments. While QD-MCTS still surpasses vanilla MCTS in some maps, the strength of sampling probably diminishes the results.

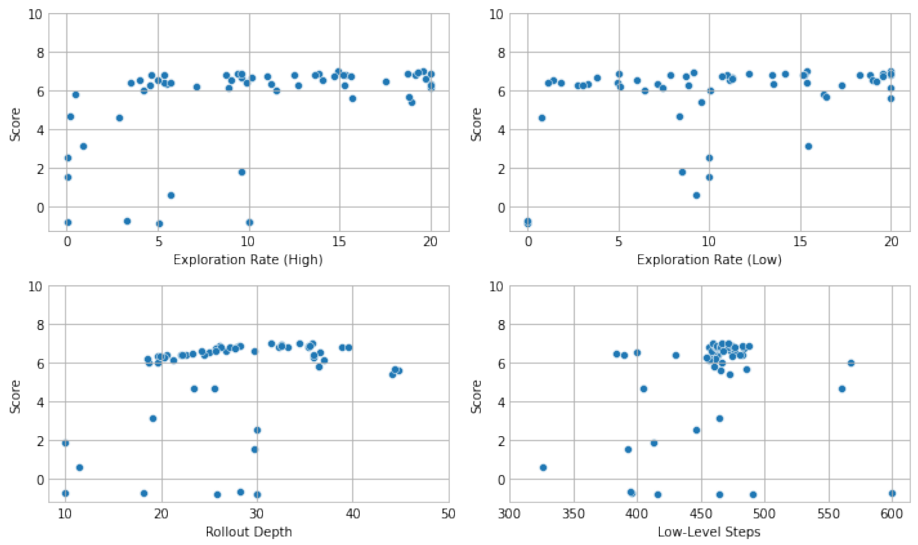


Figure 4.4: The parameter tested by the Bayesian optimization and the corresponding score for QD-MCTS. Note that the parameter choices affect each other, meaning low scores can appear everywhere if another parameter was poorly chosen.

4.3 Randomized Waypoints

The parameter optimization in Section 4.2 used maps that contained ten predefined waypoints. Because the number of waypoints affects how dense the reward landscape is, we decided to test how the algorithms behave if we change the number of waypoints.

4.3.1 Setup

Instead of manually placing waypoints on each map, we uniformly sample positions until N positions are found that do not overlap with a wall. By doing this, we can easily change the number of waypoints on each map. Additionally, the randomization allows us to test if the algorithms were overfitted to the predefined waypoints used in the optimization. We run each algorithm ten times for each of the ten maps. Note that an algorithm will never see the same constellation of waypoints, even on the same map.

For this experiment, we compare all the algorithms defined in Section 4.2. Furthermore, we also include another QD-MCTS algorithm with handpicked parameters to test if the assumption is correct that the high exploration rate from the optimization does not matter. We call this algorithm QD-MCTS-HP, and it has a high-level exploration rate of 4, a low-level exploration rate of $\sqrt{2}$, a rollout depth of 25, and the low-level search runs for 460 steps. At last, we also include the domain-dependent MCTS algorithm from (Perez et al., 2014). Note, the algorithm cannot solve environments with more than ten waypoints due to exponential growth in waypoint combinations.

4.3.2 Results

To compare the results for varying number of waypoints, we use the normalized score, which is computed by using Equation (4.1) and the maximum number of waypoints $max(w_i)$:

$$normalizedScore(i) = \frac{score(i) + 1}{max(w_i) + 1} \tag{4.2}$$

Equation (4.1) bounds are $(-1, max(w_i))$, as such Equation (4.2) ranges from 0 to 1. Using the normalized score, we can easily compare the results for varying number of waypoints distributed on the maps. Figure 4.5 shows the normalized score that the algorithms achieved, averaged over all maps.

First of all, we can see that the score increases the more waypoints are distributed on the map. This is because it becomes easier to reach more waypoints in fewer steps, the denser the waypoints are distributed. Since we did not change the map size, more waypoints result in a denser distribution. Note that the score increases for all algorithms, meaning all algorithms can handle different number of waypoints without any problems.

Another important observation is that QD-MCTS, QD-MCTS-HP and vanilla MCTS seem to behave identically. In general, these three algorithms achieve a higher score than any of the other tested algorithms. The only exception for this is the rule-based MCTS algorithm, which achieves the highest score overall. It is difficult to say how RB-MCTS would behave with more waypoints. However, based on the observations, the score would probably grow and thus still be the highest.

At last, in Section 4.2.2 we discussed that the optimized hyperparameters for QD-MCTS deviate immensely from the norm. To test if this deviation is necessary, we have chosen more reasonable parameters for QD-MCTS-HP. Since QD-MCTS and QD-MCTS-HP behave nearly identical, we can conclude that more reasonable parameters still achieve good results.

At last, it makes sense to have a closer look at how the algorithms performed on each map individually. All the results can be found in Appendix C, but overall they are very similar to what we have seen in Figure 4.5. One key observation is that vanilla MCTS slightly outperforms QD-MCTS and QD-MCTS-HP on maps with vast open areas. However, both QD-algorithms significantly outperform vanilla MCTS on the challenging maze from map 56, as Figure 4.6 shows. This is probably because vanilla MCTS heavily relies on sampling, which is less effective in mazes. Meanwhile, the subgoals from QD-MCTS enable it to explore the maze more efficiently and find more waypoints.

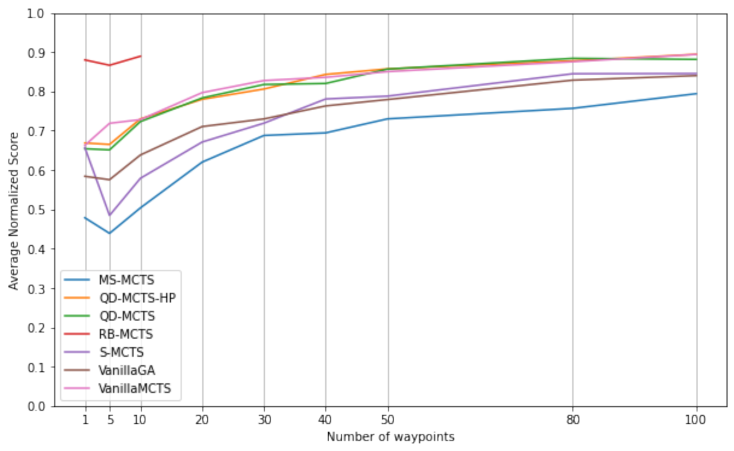


Figure 4.5: The average score that each algorithm achieved for different number of waypoints. The average is taken over all maps available in the PTSP framework.

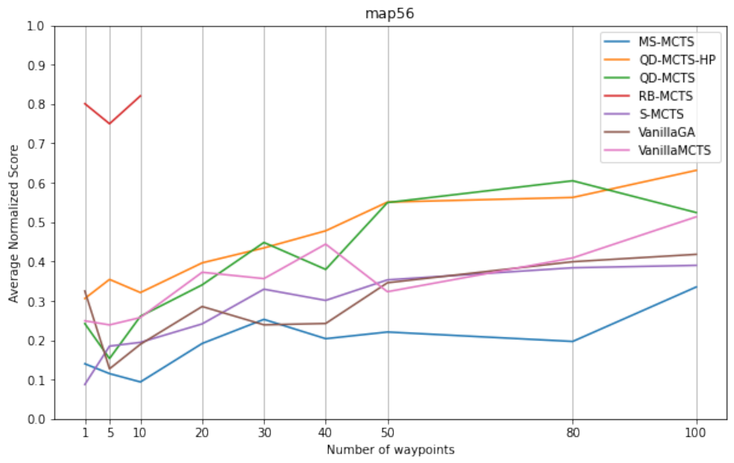


Figure 4.6: The average score that each algorithm achieved for different amounts of waypoints on map 56.

4.4 Stochastic Actions

The original S-MCTS algorithm only works in deterministic environments, this is because the low-level search in S-MCTS assumes that a macro-action will always lead to the same subgoal. Technically, QD-algorithms work in stochastic environments, and as such QD-MCTS also works in them. However, the subgoal selection algorithm from Section 3.4 uses the last state observed by a tree node to compute the novelty score. While this still makes it useable in stochastic environments, the results will rely on luck that the last observed state represents the average. Still, with this experiment, we want to test how QD-MCTS behaves in stochastic environments. Although the performance will probably be affected by not considering stochasticity, we can use it as a baseline for future improvements to the low-level search.

4.4.1 Setup

To test QD-MCTS in a stochastic environment, we use a modified version of the PTSP environment. In the modified environment, whenever an agent executes an action, it will fail with probability p and execute a random action instead. The probabilities used in this experiment are $[0.0, 0.01, 0.05, 0.1, 0.2, 0.5, 0.8, 0.9]$. We evaluate the algorithms ten times on all maps with the same predefined waypoints as used in Section 4.2.

The algorithms used for this experiment are the optimized QD-MCTS algorithm, QD-MCTS-HP, and the optimized vanilla MCTS algorithm. We have excluded S-MCTS and MS-MCTS because they are deterministic. We also excluded the vanilla genetic algorithm since it showed worse performance than vanilla MCTS, see Section 4.2.2.

4.4.2 Results

Figure 4.7 shows the normalized score that each algorithm achieved, averaged over all maps. It is easy to see that QD-MCTS handles stochastic environments quite poorly in comparison to vanilla MCTS. The same is true when looking at the results for each map individually.

As explained in Section 4.4, the low-level MCTS search presented in this thesis uses the last observed state to compute the novelty score. Thus, when selecting subgoals, the algorithm must rely on luck that the last observed state represents the average. This behaviour could be a possible explanation for why QD-MCTS has so much difficulty dealing with stochastic environments.

However, this should not be such a big problem with small probabilities. With small probabilities like 0.01, it is unlikely that the last observed state diverges immensely from the average. One observation that we have made is that QD-MCTS often misses waypoints by a tiny margin. While this also happens in deterministic environments, it happens very often in the stochastic environment. First of all, this likely happens in deterministic environments because the low-level search returns the last sampled trajectory for usage in the high-level search. Since most sampled trajectories will not be added as subgoals to the high-level search, the value estimation is essentially biased. For example, the low-level search samples a trajectory that reaches a waypoint but does not use that trajectory as a subgoal. As a result, the high-level search uses rewards from trajectories that are not reachable anymore. This effect is likely amplified by stochasticity, and as a consequence, significantly reduces the performance of QD-MCTS.

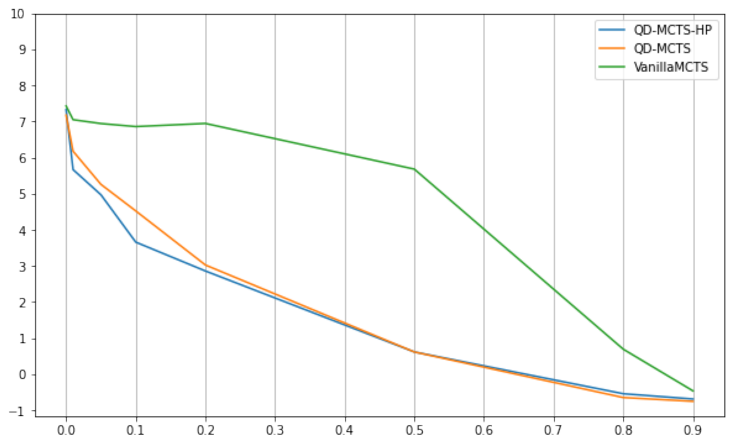


Figure 4.7: The average score that each algorithm achieved for different degrees of randomness. The average is taken over all maps available in the PTSP framework.

4.5 Search Tree Analysis

The previous experiments have shown that vanilla MCTS and QD-MCTS achieve nearly the same results. However, QD-MCTS managed to outperform all algorithms on the challenging maze map.

Figure 4.8 shows examples of how the search looks like for both algorithms. Note that the figure shows the search tree for QD-MCTS-HP, which achieves similar results as the optimized version of S-MCTS. The most significant difference is that QD-MCTS-HP explores deeper lines than the optimized QD-MCTS algorithm.

Now, while the search tree from vanilla MCTS looks at the immediate future, QD-MCTS looks much further. So, while it is true that vanilla MCTS and QD-MCTS behave similar, the way they find the solutions seem to be quite different. The only way for vanilla MCTS to find waypoints is by sampling or getting close to a waypoint. Meanwhile, QD-MCTS can find waypoints without sampling, which gives it the edge in the maze map.

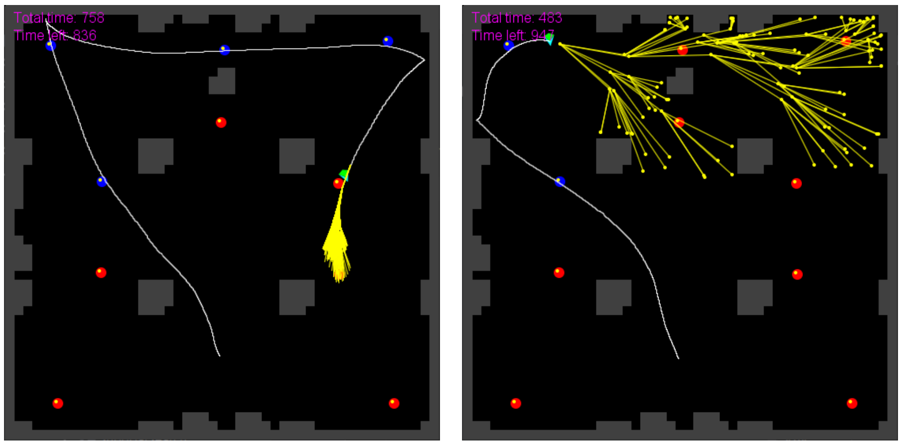


Figure 4.8: Example visualisations of the search tree for vanilla MCTS (Left) and QD-MCTS-HP (Right).

Chapter 5

Conclusion

In this chapter, we conclude this thesis. Section 5.1 and Section 5.2 address this thesis's research questions and problem statement based on the results and discussions from previous chapters. Lastly, Section 5.3 describes potential future research directions.

5.1 Research Questions

In this section, we discuss the research questions from Section 1.3 using the findings of the previous chapters.

1. *What states qualify as useful subgoals for Subgoal-MCTS?*

In Section 3.1 and Section 3.2, we have identified that the most optimal subgoals are states that lie on the optimal trajectory. However, finding these subgoals is quite difficult and time-consuming. To provide a better alternative, this thesis explored the idea of treating any state as a subgoal. Thus, instead of spending much time finding the optimal subgoals, we can find a set of states that enable S-MCTS to explore the state-space sufficiently. This idea is based on the assumption that in environments with complex action spaces, slight deviations from the optimal path often achieve similar results.

2. *What metrics can be used to detect the subgoals?*

Quality Diversity (QD) algorithms (Pugh et al., 2016) focus on finding a set of solutions that are as diverse as possible while still seeking the niche's best-performing individual. For this, the QD-algorithms make use of a behaviour function that maps a solution onto a behaviour space. By defining the behaviour function as the state reached by an action-sequence, QD-algorithms can be adapted to find a set of macro-actions that lead to novel states and are highly rewarding. As such, QD-algorithms can be used to find states that enable S-MCTS to explore the state-space.

3. *How can a search algorithm quickly detect subgoals?*

Most QD-algorithms are anytime algorithms, they can be stopped at any moment, and they will return a valid solution. This property is beneficial because the time the QD-search needs can be adapted based on the user's requirements.

4. *How can subgoal-detection be integrated into Subgoal-MCTS to improve its performance?*

In Section 3.3 we have shown that the low-level search from S-MCTS can be easily replaced with a QD-search. We have also shown in Chapter 4 that the inclusion of a QD-search improves the performance of S-MCTS significantly.

5.2 Problem Statement

The problem of this statement was formulated as follows:

How can we automatically detect subgoals in real-time for usage in Subgoal-MCTS?

By replacing the low-level search in S-MCTS with a QD-algorithm, we can enable it to find subgoals in real-time. The resulting algorithm called QD-MCTS has shown to improve the performance of S-MCTS significantly in the Physical Travelling Salesman Problem (PTSP) (Perez Liebana et al., 2012).

However, we have also shown that a vanilla MCTS algorithm can achieve similar results as QD-MCTS on most maps. It is noteworthy that QD-MCTS managed to outperform all tested algorithms on the most challenging map in PTSP. We have shown that this behaviour is likely caused by the fact that most PTSP maps have vast open areas in which sampling is more efficient than searching. In contrast, the difficult map in which QD-MCTS performed exceptionally is a maze in which sampling is nearly ineffective. These results indicate that QD-MCTS has the potential to perform very well in environments with complex action spaces where searching is unavoidable.

5.3 Future Research

We have evaluated QD-MCTS only in one environment. As such, one crucial step is to investigate how QD-MCTS performs in other environments. Additionally, although the used environment requires an agent to look far ahead, the action-space is still relatively small. Therefore, it would be interesting to see how QD-MCTS performs in environments with a large branching factor.

However, there seems to be a critical weakness in QD-MCTS that might need to be addressed before applying QD-MCTS to more complex environments. The high-level search receives trajectories from the low-level search that are no longer reachable once the low-level search is completed. The result is that the value estimation in the high-level search is biased. One possible solution is to use a QD-algorithm that can find one subgoal at a time. It might also be interesting to investigate if it is necessary to make the low-level search pausable. Maybe it is sufficient to find all subgoals immediately without returning any sampled trajectories.

In general, trying out different QD-algorithms seems rather promising. The proposed low-level MCTS search does its job. However, in sparse environments, MCTS behaves too much like a breadth-first search, limiting the length of the macro actions that can be found. Additionally, the proposed low-level MCTS search does not explicitly search for diverse macro-actions, which is likely wasting computation time on redundant trajectories.

References

Aubret, A., Matignon, L., & Hassas, S. (2019). A survey on intrinsic motivation in reinforcement learning. *CoRR*, *abs/1908.06976*. Retrieved from `http://arxiv.org/abs/1908.06976`

Baier, H., & Kaisers, M. (2021, July). Novelty and MCTS. In *Proceedings of the 1st Evolutionary Reinforcement Learning Workshop at GECCO 2021.*

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., ... Colton, S. (2012, 03). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4:1*, 1-43. doi: 10.1109/TCIAIG.2012.2186810

Campbell, M., Hoane, A., & Hsu, F.-h. (2002, 01). Deep Blue. *Artificial Intelligence*, *134*, 57-83. doi: 10.1016/S0004-3702(01)00129-1

Chaslot, G., Winands, M. H., Herik, H. J. V. D., Uiterwijk, J. W. H. M., & Bouzy, B. (2008, 11). Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, *04*, 343-357. doi: 10.1142/S1793005708001094

Coulom, R. (2007). Efficient selectivity and backup operators in monte-carlo tree search. In H. J. van den Herik, P. Ciancarini, & H. H. L. M. J. Donkers (Eds.), *Computers and Games* (pp. 72–83). Berlin, Heidelberg: Springer Berlin Heidelberg.

Dasgupta, D., & Michalewicz, Z. (1997). Evolutionary algorithms — an overview. In D. Dasgupta & Z. Michalewicz (Eds.), *Evolutionary Algorithms in Engineering Applications* (pp. 3–28). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from `https://doi.org/10.1007/978-3-662-03423-1_1` doi: 10.1007/978-3-662-03423-1_1

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., & Clune, J. (2019). Go-explore: a new approach for hard-exploration problems. *CoRR*, *abs/1901.10995*. Retrieved from `http://arxiv.org/abs/1901.10995`

Gabor, T., Peter, J., Phan, T., Meyer, C., & Linnhoff-Popien, C. (2019, 7). Subgoal-based temporal abstraction in monte-carlo tree search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19* (pp. 5562–5568). International Joint Conferences on Artificial Intelligence Organization. Retrieved from `https://doi.org/10.24963/ijcai.2019/772` doi: 10.24963/ijcai.2019/772

Gaina, R. D., Devlin, S., Lucas, S. M., & Perez-Liebana, D. (2020). *Rolling horizon evolutionary algorithms for general video game playing.*

Geffner, T., & Geffner, H. (2021, Jun.). Width-based planning for general video-game playing. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *11*(1), 23-29. Retrieved from `https://ojs.aaai.org/index.php/AIIDE/article/view/12786`

Genesereth, M., Love, N., & Pell, B. (2005, Jun.). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, *26*(2), 62. Retrieved from `https://ojs.aaai.org/index.php/aimagazine/article/view/1813` doi: 10.1609/aimag.v26i2.1813

Gregor, K., Rezende, D. J., & Wierstra, D. (2016). Variational intrinsic control. *CoRR*, *abs/1611.07507*. Retrieved from `http://arxiv.org/abs/1611.07507`

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293-326. doi: https://doi.org/10.1016/0004-3702(75)90019-3

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. In J. Fürnkranz, T. Scheffer, & M. Spiliopoulou (Eds.), *Machine learning: Ecml 2006* (pp. 282–293). Berlin, Heidelberg: Springer Berlin Heidelberg.

Lehman, J., & Stanley, K. (2008, 01). Exploiting open-endedness to solve problems through the search for novelty. *Artificial Life - ALIFE*.

Lehman, J., & Stanley, K. (2011a, 06). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, *19*, 189-223. doi: 10.1162/EVCO_a_00025

Lehman, J., & Stanley, K. O. (2011b). Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (p. 211–218). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2001576.2001606` doi: 10.1145/2001576.2001606

McGovern, A., & Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning* (p. 361–368). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Pepels, T., & Winands, M. H. M. (2012). Enhancements for monte-carlo tree search in Ms Pac-Man. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)* (p. 265-272). doi: 10.1109/CIG.2012.6374165

Perez, D., Powley, E. J., Whitehouse, D., Rohlfshagen, P., Samothrakis, S., Cowling, P. I., & Lucas, S. M. (2014). Solving the physical traveling salesman problem: Tree search and macro actions. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), 31-45. doi: 10.1109/TCIAIG.2013.2263884

Perez Liebana, D., Rohlfshagen, P., & Lucas, S. (2012, 06). The physical travelling salesman problem: WCCI 2012 Competition. In *IEEE Congress on Evolutionary Computation* (Vol. 1). doi: 10.1109/CEC.2012.6256440

Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S. M., Couëtoux, A., … Thompson, T. (2016). The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, *8*(3), 229-243. doi: 10.1109/TCIAIG.2015.2402393

Pugh, J. K., Soros, L. B., & Stanley, K. O. (2016). Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, *3*, 40. Retrieved from `https://www.frontiersin.org/article/10.3389/frobt.2016.00040` doi: 10.3389/frobt.2016.00040

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., … et al. (2020, Dec). Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609. Retrieved from `http://dx.doi.org/10.1038/s41586-020-03051-4` doi: 10.1038/s41586-020-03051-4

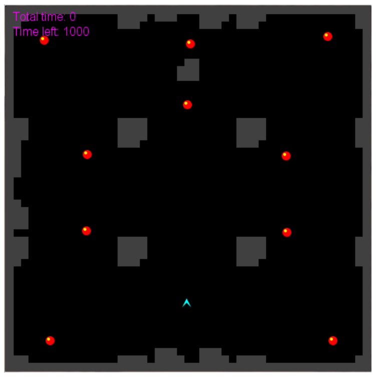
Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., … Hassabis, D. (2016, 01). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*, 484-489. doi: 10.1038/nature16961

Snoek, J., Larochelle, H., & Adams, R. P. (2012). *Practical bayesian optimization of machine learning algorithms.*

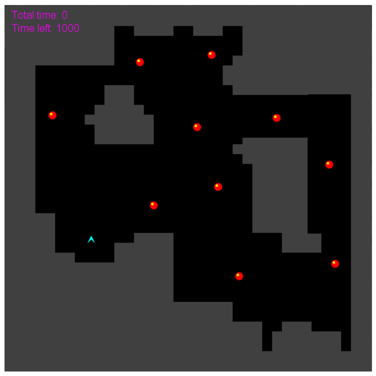
Soemers, D. J. N. J., Sironi, C. F., Schuster, T., & Winands, M. H. M. (2016). Enhancements for real-time monte-carlo tree search in general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)* (p. 1-8). doi: 10.1109/CIG.2016.7860448

Appendix

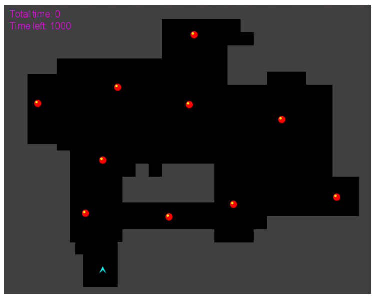
A PTSP Maps



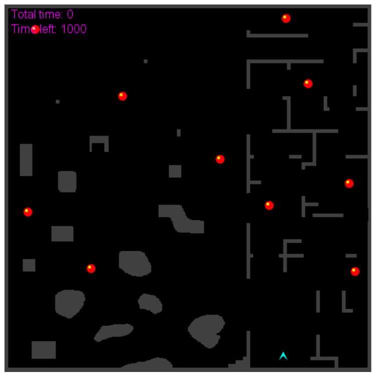
Map 01



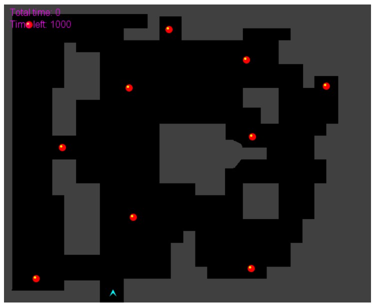
Map 02



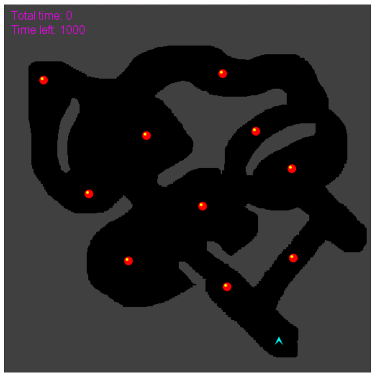
Map 08



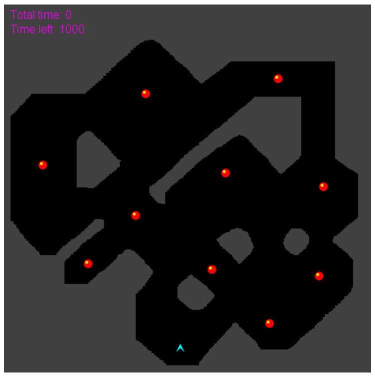
Map 19



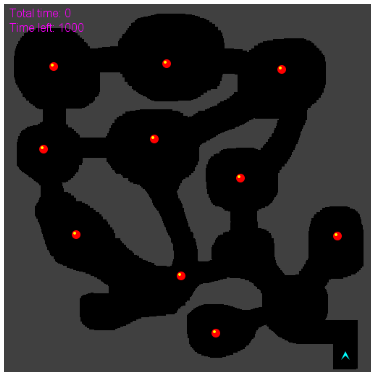
Map 24



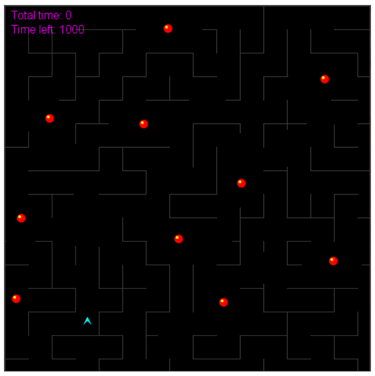
Map 35



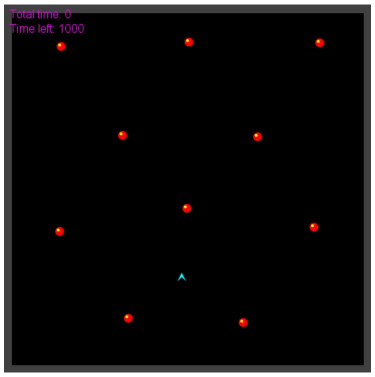
Map 40



Map 45



Map 56



Map 61

B Results Hyperparameter Optimization

Table 1: Results of the bayesian optimization for all algorithms

	Parameter	Bound	Result	Score
QD-MCTS	Exploration Rate (High)	(0.01, 20)	14.93	7.00
	Exploration Rate (Low)	(0.01, 20)	15.34	
	Low-Level Steps	(300, 600)	460	
	Rollout Depth	(10, 50)	32	
S-MCTS	Predicate Grid Size	(5, 30)	29	5.06
	Exploration Rate (High)	(0.01, 20)	13.81	
	Low-Level Steps	(300, 600)	513	
	Rollout Depth	(10, 50)	40	
MS-MCTS	Predicate Grid Size	(5, 30)	5	4.53
	Exploration Rate (High)	(0.01, 20)	9.36	
	Low-Level Steps	(300, 600)	560	
	Rollout Depth	(10, 50)	22	
VanillaMCTS	Exploration Rate	(0.01, 20)	7.80	6.83
	Rollout Depth	(10, 50)	35	
VanillaGA	Genome Length	(10, 50)	35	5.77
	Population Size	(1, 200)	151	
	Mutation Rate	(0.01, 1)	0.25	

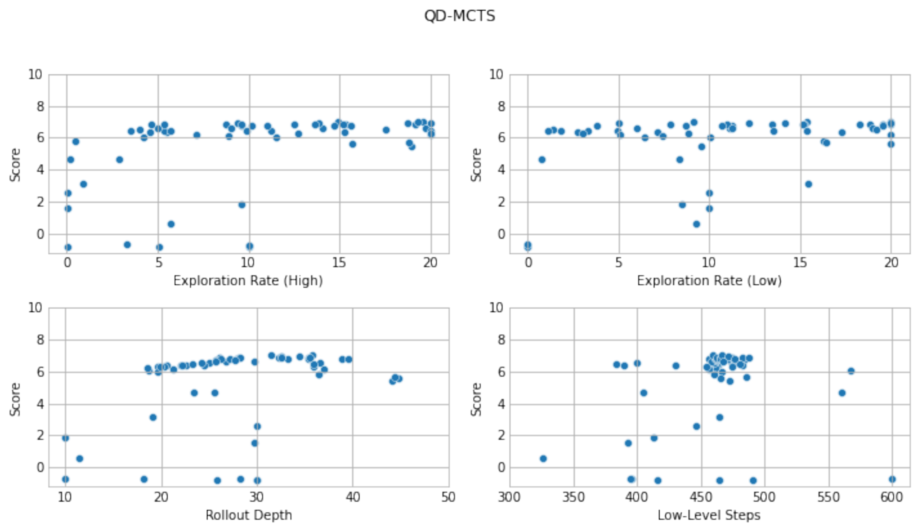


Figure 1: The parameter tested by the Bayesian optimization and the corresponding score for QD-MCTS.

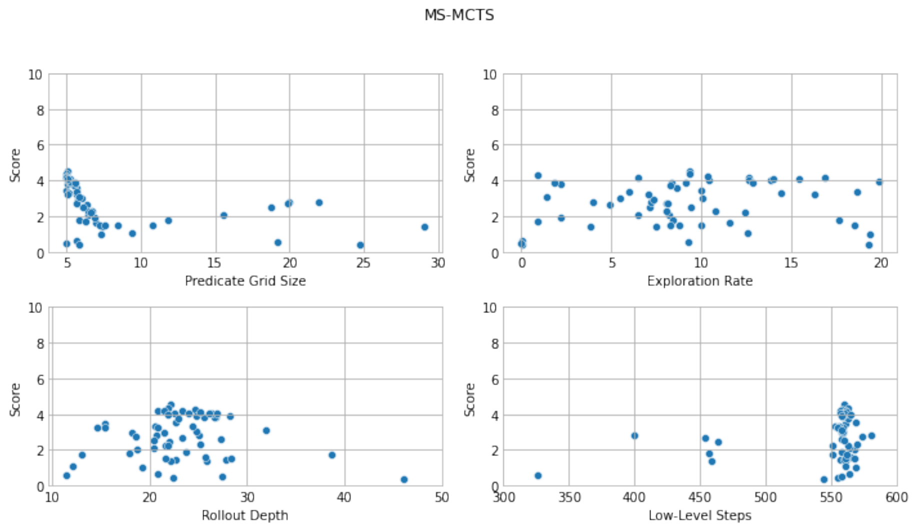


Figure 2: The parameter tested by the Bayesian optimization and the corresponding score for MS-MCTS.

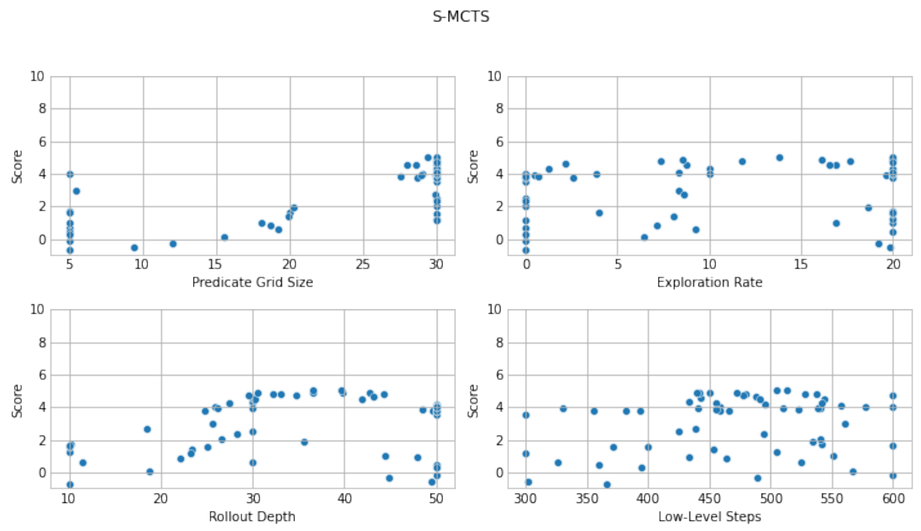


Figure 3: The parameter tested by the Bayesian optimization and the corresponding score for S-MCTS.

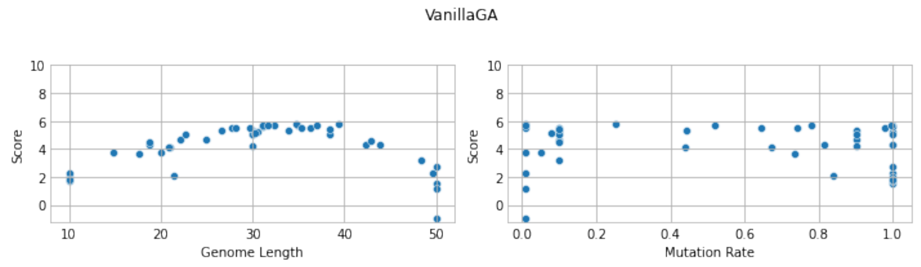


Figure 4: The parameter tested by the Bayesian optimization and the corresponding score for the vanilla genetic algorithm.

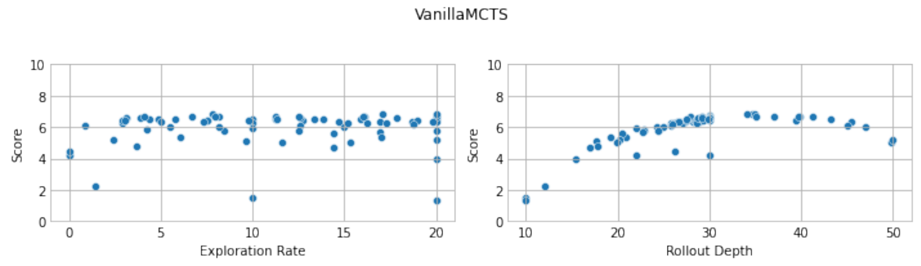


Figure 5: The parameter tested by the Bayesian optimization and the corresponding score for vanilla MCTS.

C Results Randomized Waypoints

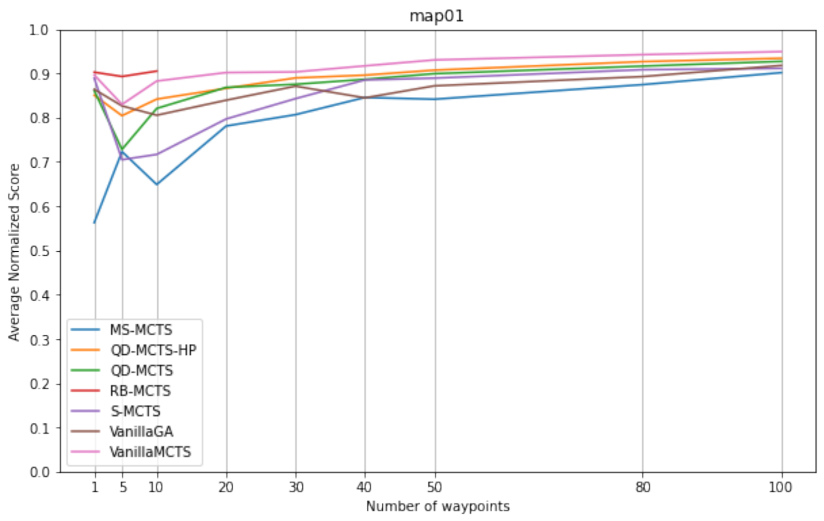


Figure 6: The average score that each algorithm achieved for different amounts of waypoints.

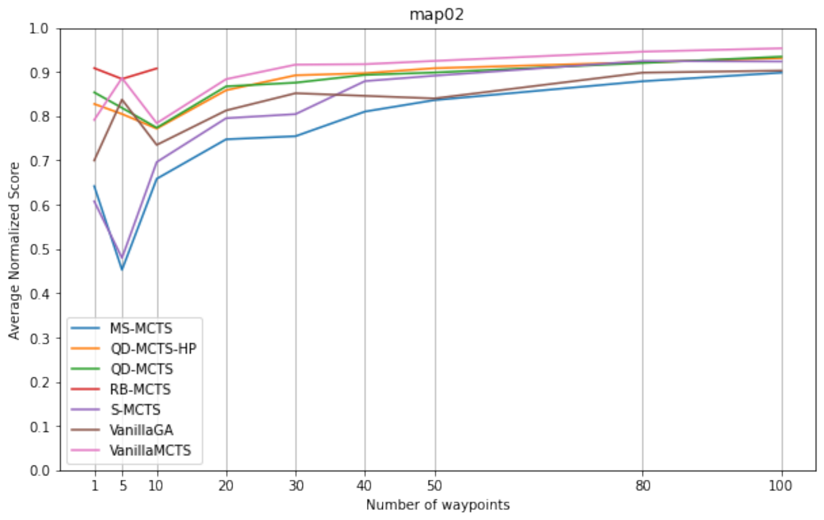


Figure 7: The average score that each algorithm achieved for different amounts of waypoints.

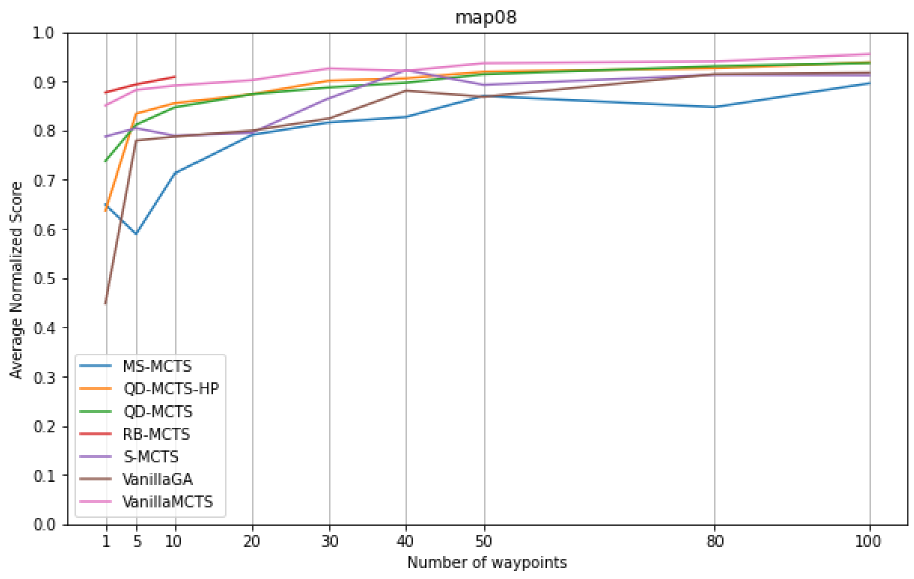


Figure 8: The average score that each algorithm achieved for different amounts of waypoints.

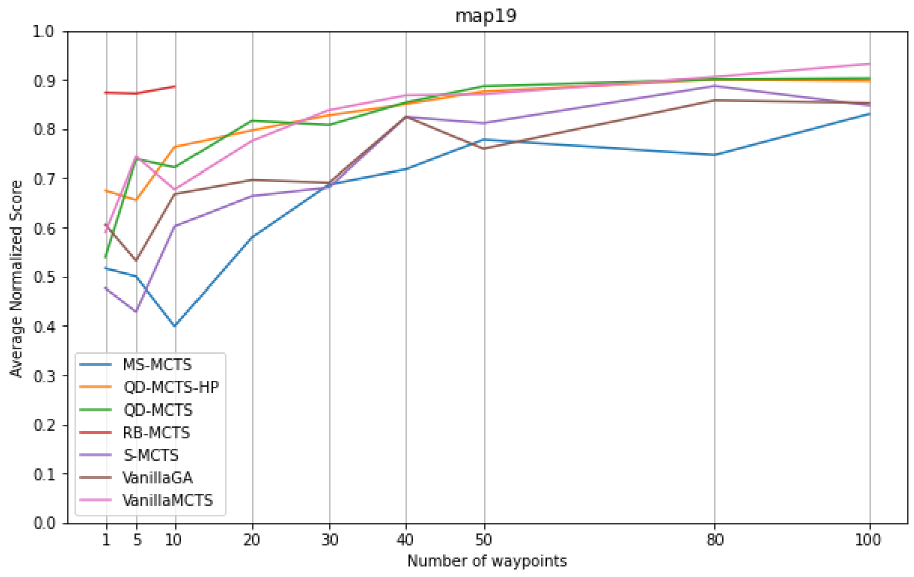


Figure 9: The average score that each algorithm achieved for different amounts of waypoints.

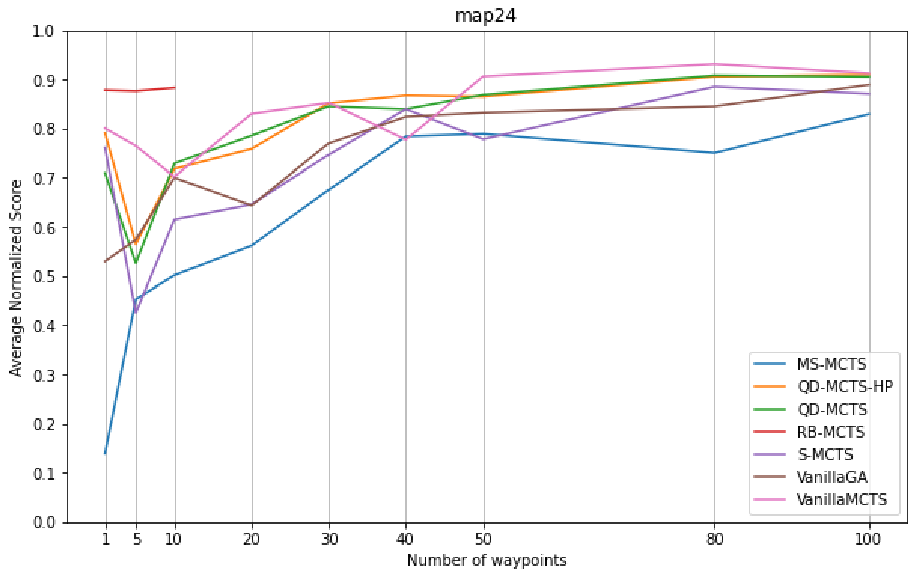


Figure 10: The average score that each algorithm achieved for different amounts of waypoints.

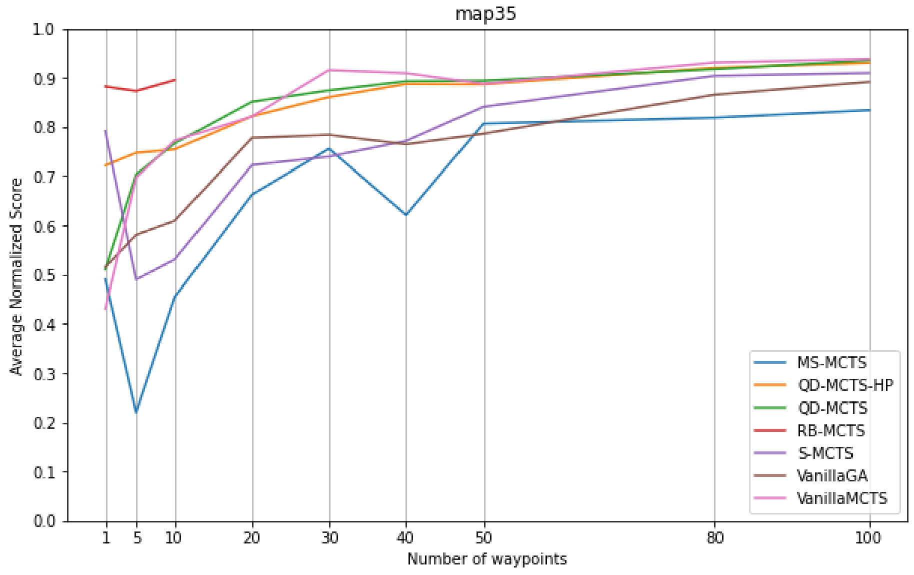


Figure 11: The average score that each algorithm achieved for different amounts of waypoints.

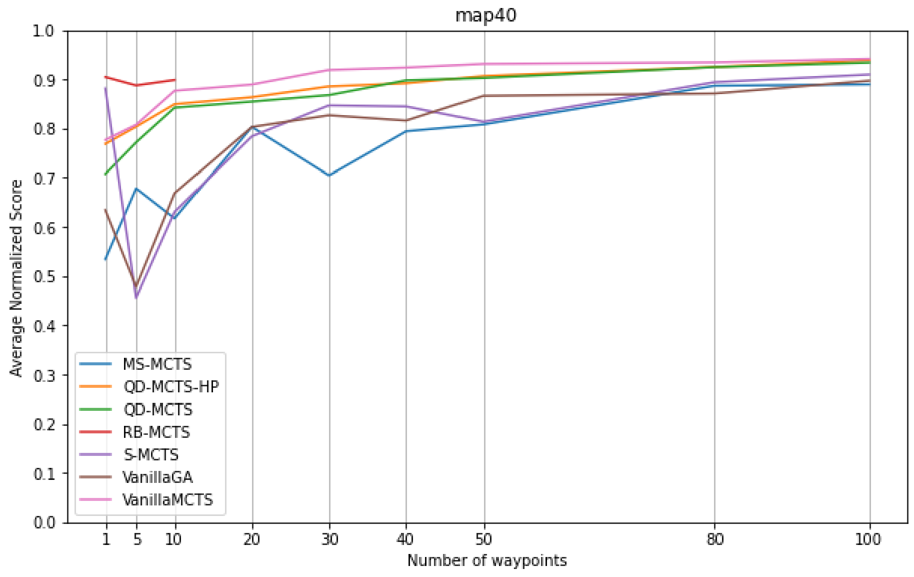


Figure 12: The average score that each algorithm achieved for different amounts of waypoints.

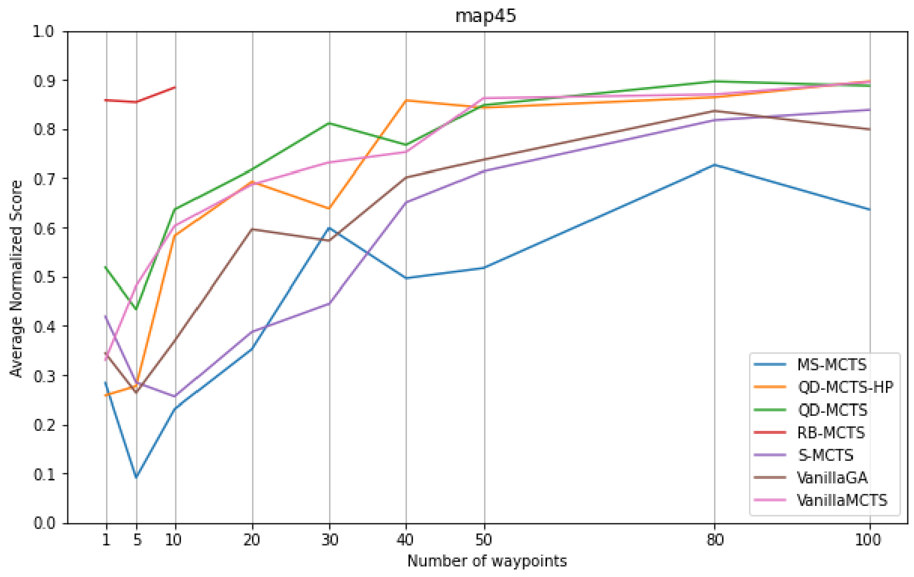


Figure 13: The average score that each algorithm achieved for different amounts of waypoints.

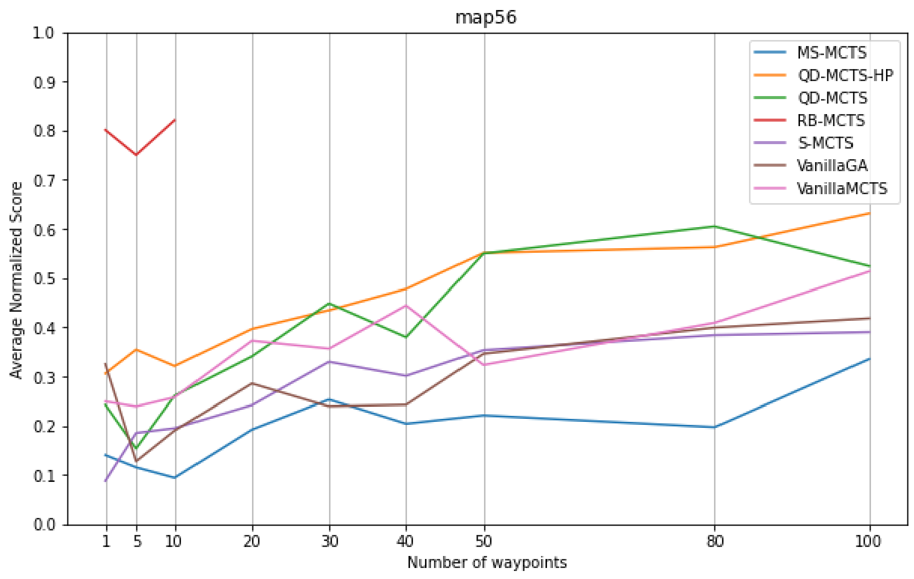


Figure 14: The average score that each algorithm achieved for different amounts of waypoints.

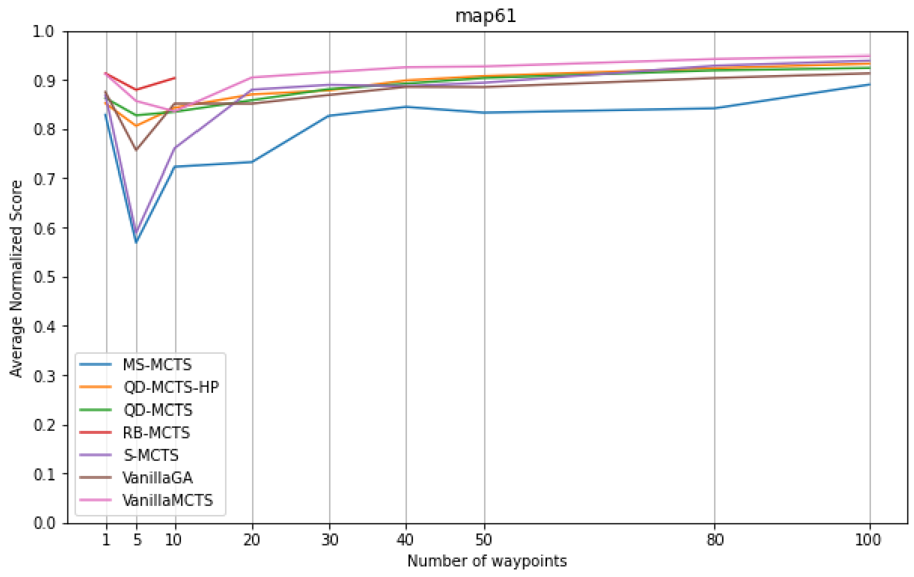


Figure 15: The average score that each algorithm achieved for different amounts of waypoints.