Karlsruhe Institute of Technology

Decentralized Planning of Macro-Actions for Cooperative Automated Vehicles with Hierarchical Monte Carlo Tree Search

Master's Thesis of

Chenyang Zhou

Institute of Measurement and Control
Karlsruhe Institute of Technology

Reviewer:	Prof. Dr.-Ing. Christoph Stiller
Advisors:	Karl Kurzer, M.Sc.
	Maximilian Naumann, M.Sc.

Karlsruhe, Februrary 2018

KIT – The Research University in the Helmholtz Association

www.kit.edu

Declaration / Erklärung

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannten Hilfsmittel selbständig angefertigt, alle benutzten Hilfsmittel vollständig angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Chenyang Zhou
Karlsruhe, 28.02.2018

Abstract

Although motion planning in automated driving has been studied for a long time, today's automated vehicles still lack the ability to cooperate implicitly with others. This thesis presents a Monte Carlo Tree Search (MCTS) based approach: Decentralized Hierarhical Monte Carlo Tree Search (DecH-MCTS), to achieve decentralized planning of macro-actions for cooperative automated vehicles. Based on the assumption of perfect perception and cooperative agents, the agent reward function combines both ego and others' reward so that the resulting behaviors can be considered cooperative. Because the performance of MCTS is dominated by its effective search depth, macro-actions, which allow temporal extension from one to multiple time steps, are integrated into the MCTS to increase the effective search depth. Inspired by hierarchical reinforcement learning methods, we propose the notion of a hierarchically bounded return for the integration process. Distributed reinforcement learning as well as Decoupled Upper Confidence bound Tree (DUCT) (a variant of MCTS) methods are considered to realize decentralized decision making in the multi-agent system. Without hard-coded policies for macro-actions, the algorithm simultaneously learns policies over and within macro-actions.

The proposed method is evaluated under several conflict scenarios based on the comparison with the classical MCTS, which shows that our algorithm outperforms the classical version in terms of faster and better convergence. Its robustness in heterogeneous environments is examined later and the result demonstrates that DecH-MCTS is able to generate feasible solutions when encountering unknown drivers.

Contents

Abstract v

Acronyms ix

1. Introduction 1
- 1.1. Motivation 1
- 1.2. Goal Description and Limitations 1
- 1.3. Outline 2
- 1.4. Presupposed Knowledge 2

2. Fundamentals and Related Work 3
- 2.1. Markov Decision Process 3
  - 2.1.1. Classical MDP 3
  - 2.1.2. Semi-MDP 4
  - 2.1.3. Multi-Agent (Semi-)MDP 5
- 2.2. Reinforcement Learning and Planning 6
  - 2.2.1. Simulation-based Search 6
  - 2.2.2. Hierarchical Reinforcement Learning 7
  - 2.2.3. Monte Carlo Tree Search 9
  - 2.2.4. Monte Carlo Tree Search in Simultaneous Move Games 10
- 2.3. Related Work 11
  - 2.3.1. Cooperative Automated Driving 11
  - 2.3.2. Monte Carlo Tree Search with Macro-Actions 13
  - 2.3.3. Macro-Actions in Multi-Agent Systems 14

3. Underlying Conditions and Preliminaries 15
- 3.1. Underlying Conditions 15
- 3.2. Preliminaries 15
  - 3.2.1. Nomenclature 15

4. Concepts 17
- 4.1. Planning with Primitive Motions 17
  - 4.1.1. Action Space 17
  - 4.1.2. Reward Function 18
  - 4.1.3. Monte Carlo Learning 20
  - 4.1.4. Decentralization 20
- 4.2. Planning with Macro-Actions 21
  - 4.2.1. Design of Macro-Actions 21
  - 4.2.2. Termination Scheme 23
  - 4.2.3. Reward Function 23
  - 4.2.4. Learning with Hierarchically Bounded Return 23

5. Implementation 27
5.1. Overview 27
5.2. Tree Policy 27
5.2.1. Selection 29
5.2.2. Expansion 30
5.3. Simulation Policy 31
5.4. Backpropagation 31
5.4.1. Reward Collection 31
5.4.2. Computation of Hierarchically Bounded Return 32
5.5. Final Selection and Execution 33

6. Evaluation 35
6.1. Test Scenarios 35
6.1.1. Free Drive 35
6.1.2. Merge 35
6.1.3. Double Merge 36
6.1.4. Overtaking 36
6.1.5. Bottleneck 36
6.1.6. Open Loop 37
6.2. General Applicability 37
6.3. Performance Analysis 38
6.3.1. Flexible Learning of Macro-Actions 38
6.3.2. Faster Learning than Classical MCTS 41
6.3.3. Robustness 45

7. Conclusion and Future Work 51
7.1. Conclusion 51
7.2. Future Work 51

A. Appendix 53
A.1. Parameter Settings 53
A.2. Scenario Descriptions 54
A.3. Solutions found by DecH-MCTS 55
A.4. Computation Time 55
A.5. Original Simulation Result in the Evaluation of Learning Speed 55

List of Figures 67

List of Tables 69

Bibliography 71

Acronyms

DecH-MCTS	Decentralized Hierarhical Monte Carlo Tree Search
Dec-SMDP	Decentralized Semi-MDP
DQN	Deep Q-Network
DUCT	Decoupled Upper Confidence bound Tree
GLIE	Greedy in the Limit with Infinite Exploration
HRL	Hierarchical Reinforcement Learning
ISMCTS	Information-Set MCTS
MAMDP	Multi-Agent MDP
MAXQ	MAXQ Value Function Decomposition
MCTS	Monte Carlo Tree Search
MDP	Markove Decision Process
OL-MCTS	Option Learning MCTS
PBRS	Potential Based Reward Shaping
POMDP	Partially Observable MDP
RL	Reinforcement Learning
SMDP	Semi-MDP
UCB1	Upper Confidence Bound
UCT	Upper Confidence bound Tree

1. Introduction

1.1. Motivation

Compared with the traditional maneuver planning methods, human drivers take other drivers' reactions into consideration, enabling them to generate cooperative plans. Although a variety of cooperative planning approaches for vehicles have been proposed recently which take the interdependence of one's own action and the others' actions into account, it is still difficult to make this reality considering the required communication between vehicles and the respective computational demand.

Taking the existence of human drivers into consideration, V2X communication should not be compulsory to achieve cooperative planning [NS17, NOB$^+$17]. Additionally, to realize online planning in multiple vehicles is computationally demanding, because the number of possible outcomes grows exponentially as the number of agents increases and plans for longer time horizons.

MCTS has shown promising performance on multiple occasions facing problems of this kind. The most popular example is AlphaGo which reaches super-human performance in the game of Go [SHM$^+$16a, SSS$^+$17a]. It is shown that the performance of MCTS is dominated by its effective search depth [KMN99]. Considering that multi-agent problems have an inherently large branching factor, temporally extendable actions, in other words, macro-actions demonstrate promising performance in increasing the search depth [SPS99]. As a result, the interplay of MCTS and macro-actions is of great potential in solving the above mentioned challenges.

1.2. Goal Description and Limitations

To address the problem of decentralized decision making and planning for longer time horizons, this work aims to realize the classical MCTS for decentralized maneuver planning and to integrate the macro-actions to increase the effective search depth.

Contribution

We propose the algorithm *Decentralized Hierarchical Monte Carlo Tree Search* (DecH-MCTS) to achieve planning of macro-actions in a decentralized manner. Within this algorithm the contributions are:

- Four types of macro-actions are designed in the form of a hierarchical action graph;

- Policies over and within the macro-actions can be online simultaneously learned so that macro-actions are adaptively generated according to the environment;
- The comparison shows that DecH-MCTS outperforms classical MCTS in terms of learning speed and optimality;
- Experiments in heterogeneous environment demonstrate great robustness of DecH-MCTS.

Limitations

The action space of the primitive actions is strongly discretized: only *acceleration*, *deceleration*, *do nothing*, *left change* and *right change* with fixed trajectories are available. Besides, the algorithm lacks online adaptive estimation of other agent agents, which is useful for improving the robustness in highly complex scenarios. Lastly, the action space after choosing a macro-action is limited to the macro-action's child actions (see Sec. 4.2.1), which results in the recursive optimality of the generated plan instead of global optimality.

1.3. Outline

This thesis is organized as follows: Chapter 2 gives an overview of the theoretical fundamentals about the Markove Decision Process (MDP), reinforcement learning and MCTS. Current research on cooperative planning in automated driving area is also presented later. The problem formulation and terminology is introduced in Chapter 3. The fundamental concepts of our algorithm are presented in Chapter 4, followed by its implementation details in Chapter 5. Lastly, the algorithm is evaluated under multiple scenarios in Chapter 6.

1.4. Presupposed Knowledge

The knowledge on classical tree search algorithms and the basics of reinforcement learning and planning is helpful for understanding derivations of formulas in this thesis.

2. Fundamentals and Related Work

This section provides the theoretical background of the proposed method. Firstly, the markov decision process and its variants are presented, which serves as the basis for the formulation of our problem as a Decentralized Semi-MDP (Dec-SMDP). The basic ideas and classical methods of reinforcement learning are introduced, which provides the theoretical support for our proposed concepts in Chapter 4. MCTS is then illustrated as well as its relevant variants. The practices in cooperative automated driving as well as the planning of macro-actions are presented later.

2.1. Markov Decision Process

Markov Decision Process (MDP) is the most widely used framework to model a decision process of an agent in a fully observable environment. The basic assumption is that the next state is fully dependent on the present state and independent of all the past states, which can be formulated in 2.1.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, \dots, S_t) \tag{2.1}$$

2.1.1. Classical MDP

Classical MDP can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ and a policy π, where $\mathcal{S}$ is the finite state space of the agent, $\mathcal{A}$ represents the finite action space, T is the transition probability function $p(s'|s,a)$ which specifies the probability of the transition from state s to state s' under action $a \in \mathcal{A}$. $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function with $r(s, s', a)$ representing the value after the action a is executed. $\gamma \in [0, 1]$ is a discount factor which controls the influence of future rewards on the current state value. It can be seen as how shortsighted the agent is: the closer gamma is to 1, the more shortsighted the agent behaves.

Based on the reward and discount factor, the *return* G_t at time step t is defined as the discounted cumulative reward starting from current step 2.2:

$$G_t = r_{t+1} + \gamma r_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^{k-1} r_{t+k}(s, s', a) \tag{2.2}$$

A policy π is used to guide the agent how to make decisions at current state, i.e. a distribution over all possible actions a given state s_t, as Eq. 2.3 shows.

$$\pi(a|s) = p(A_t = a|S_t = s) \tag{2.3}$$

The state value function $V^\pi(s)$ is defined to describe the "quality" of being in state s under policy π. It is calculated by the expected return from this state, which can be written as Eq. 2.4.

$$\begin{aligned} V^\pi(s) &= E[G_t|S_t = s] \\ &= E[r_{t+1} + \gamma V^\pi(S_{t+1})|S_t = s] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)[r(s,s',a) + \gamma \sum_{s'} p(s'|s,a)V^\pi(s')], \end{aligned} \tag{2.4}$$

Similarly, the state-action value function $Q^\pi(s,a)$ is the expected return from state s, taking action a under policy π, as Eq. 2.5 shows.

$$\begin{aligned} Q^\pi(s,a) &= E[G_t|S_t = s, A_t = a] \\ &= r(s,s',a) + \gamma \sum_{s'} p(s'|s,a)V^\pi(s') \\ &= r(s,s',a) + \gamma \sum_{s'} p(s'|s,a) \sum_{a'} \pi(a'|s')Q^\pi(s',a'), \end{aligned} \tag{2.5}$$

The state value function and the state-action value function are dependent on the taken policy. Since the agent wants to maximize its long-term utility (expected return), the optimal value functions $V^*(s)$, $Q^*(s,a)$ are the functions with maximum value among all policies 2.6:

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ Q^*(s,a) &= \max_\pi Q^\pi(s,a) \end{aligned} \tag{2.6}$$

We can obtain the optimal policy by maximizing over $Q^*(s,a)$, i.e.,

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmin}_{a\in\mathcal{A}} Q^*(s,a) \\ 0 & \text{otherwise} \end{cases}, \tag{2.7}$$

which means that the optimal policy is obtained once $Q^*(s,a)$ is known. The problem of obtaining $Q^*(s,a)$ is intensively studied in the area of reinforcement learning illustrated in Sec. 2.2.

2.1.2. Semi-MDP

By extending the action from one time step (primitive action) to multiple time steps (macro-action), the MDP is transformed into the Semi-MDP (SMDP), which was firstly defined by [BD95]. For example, a primitive action can be *move left* or *move right*, a macro-action can be *go to the door* that involves multiple calls of primitive actions. Suppose that under policy $\mu(a|s)$, an action a is executed for τ time steps and the state is transferred from s to s', the transition probability is written as $p(s',\tau|s,a)$. The state and state-action value functions for an SMDP are generalized as Eq. 2.8.

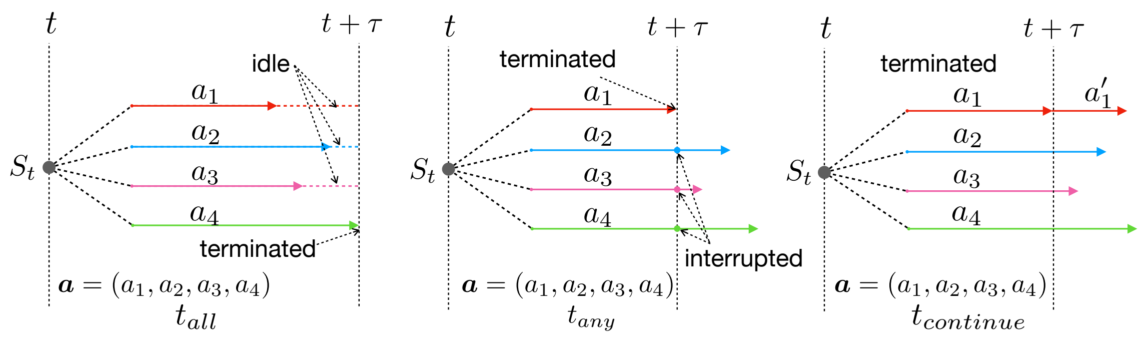


Figure 2.1.: Three termination schemes [RM03]: in t_{all}, a_1, a_2 and a_3 have shorter durations and are kept idle to wait for the termination of a_4; t_{any} interrupts all other unfinished macro-actions a_2, a_3 and a_4 when a_1 firstly terminated; in $t_{continue}$, the termination and selection of the next action are independent.

$$\begin{aligned} V^\mu(s) &= \sum_{a\in\mathcal{A}} \mu(a|s)[r(s,s',a) + \gamma^\tau V^\mu(S_{t+\tau})] \\ Q^\mu(s,a) &= r(s,s',a) + \sum_{s',\tau} \gamma^\tau p(s',\tau|s,a) \sum_{a'} \mu(a'|s')Q^\mu(s',a') \end{aligned} \tag{2.8}$$

2.1.3. Multi-Agent (Semi-)MDP

Extending MDP from the single agent domain to the multi-agent domain, we obtain the Multi-Agent MDP (MAMDP), which can be defined by the tuple $\langle \Upsilon, \mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma \rangle$. Υ is the finite set of agents indexed by $i \in 1, 2, \ldots, n$, with each having an action space $\mathcal{A}^{(i)}$. $\mathcal{A} = \times\mathcal{A}^{(i)}$ represents the joint action space of Υ. $\mathcal{S} = \times\mathcal{S}^{(i)}$ represents the joint state space of Υ with $\mathcal{S}^{(i)}$ being the finite state space of one agent. $\mathcal{T}$ is the transition probability function $p(s'|s,\boldsymbol{a})$ which specifies the probability of the transition from state s to state s' under the joint action $\boldsymbol{a}$ formulated by each agent's choice. $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function with $r^{(i)}(s, s', \boldsymbol{a})$ representing the reward after the joint action $\boldsymbol{a}$ is executed. γ is the discount factor (usually constant for each agent) and has the same definition in the classical MDP.

The solution to the MAMDP is the joint policy $\boldsymbol{\pi} = \langle \pi^{(1)}, \ldots, \pi^{(n)} \rangle$, where $\pi^{(i)}$ denotes the individual policy for each agent. By replacing the reward of the action a in MDP (Eq. 2.4 and Eq. 2.5) with the reward for the joint action $\boldsymbol{a}$, we get the state value function and the state-action value function for the MAMDP.

Taking the macro-actions into consideration, the MAMDP can be generalized into Semi-MAMDP. Since the macro-actions have different durations, the termination of the joint action $\boldsymbol{a}$ depends on the pre-defined termination scheme according to [RM03] and Fig. 2.1:

- t_{all} keeps some agents idle to wait for others finishing their actions. The decision epoch takes place when all agents terminate their actions.
- t_{any} simply interrupts all macro-actions being executed when any one finishes its own action,
- $t_{continue}$ allows asynchronous decision making. Each agent independently decides its next action once it terminates its current action.

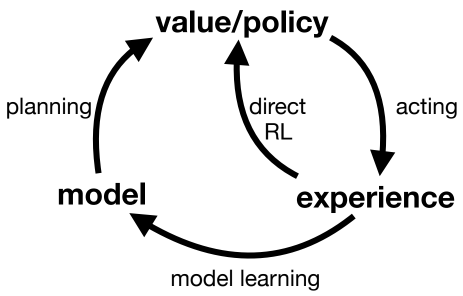


Figure 2.2.: Learning and Planning [KLM96]: the agent acts in the environment to collect experience. Direct RL (model-free RL) methods learn policies directly while, while model-based methods approximate a model of the environment and finishes the learning and planning based on the approximated model.

Clearly, the first two schemes t_{all} and t_{any} require a forced synchronization of the decision epochs and can be only realized in a centralized way, while the $t_{continue}$ allows decentralized asynchronous decision.

2.2. Reinforcement Learning and Planning

According to [KLM96], Reinforcement Learning (RL) solves the problem that an agent learns to maximize the long-term utility through the interaction with the world. The interaction process is often modeled as an MDP as Sec. 2.1.1 describes. RL provides many practical methods to solve the MDP. For details please refer to the book [SB16].

One popular classification criteria of these methods is whether the outside world is explicitly modeled: *model-based* or *model-free* learning. As Fig. 2.2 shows, model-free learning methods directly learn the policy without modeling the world, such as temporal difference (TD) learning [BSA83] and Q-learning [Wat89]. Model-based learning methods firstly approximate the world with a model, i.e., the transition matrix $\mathcal{T}$ mentioned in Sec. 2.1.1, and plan the behavior according to the model, like the Dyna algorithm [Sut91]. These two kinds of methods are closely related with each other and can be used simultaneously in practice.

2.2.1. Simulation-based Search

Simulation-based search methods can be viewed as an example of the combination of direct RL and planning in Fig. 2.2. However, instead of approximating a model from the experience, a simulator is given and the agent learns from the simulated experiences starting from the current state s_t. Using the simulation policy π, the agent simulates M episodes from s_t to s_T: $\{s_t, a_t^m, r_{t+1}^m, s_{t+2}^m, \ldots, s_T^m\}_{m=1}^M$ and builds a search tree which contains the visited states and actions in its nodes. Based on the Monte-Carlo evaluation, which is the mean value of the returns from these M episodes, the state-action value $Q(s,a)$ is approximated with Eq.2.9:

$$Q^\pi(s,a) = \frac{1}{N(s,a)} \sum_{m=1}^{M} \sum_{u=t}^{T} 1(s_u = s, a_u = a) G_u, \tag{2.9}$$

where $1(s_u = s, a_u = a)$ is a function which returns 1 at $s_u = s, a_u = a$ and 0 otherwise.

After the evaluation, the simulation policy π is improved because we get a better approximation of the Q-values. The next K simulations can be executed by maximizing over the approximated Q-values, i.e., $a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a)$. This procedure of simulation and evaluation can be repeated and the learned Q-values are proven to converge to the optimal value Q^*.

2.2.2. Hierarchical Reinforcement Learning

To address the problem of combinatoric explosion at longer horizons, the learning process of the policies is divided into a hierarchical structure and called Hierarchical Reinforcement Learning (HRL). Intuitively, the learned policy by HRL is also represented by a hierarchical structure $\pi = \{\pi_0, \pi_1, \dots\}$.

The temporal abstraction is one major part of HRL. Here the temporally extendable actions (or macro-actions) based methods are intensively studied. Based on the SMDP (see Section 2.1.2), there are three popular learning frameworks in this area: *Option* [SPS99], *HAM* (Hierarchical Abstract Machine) [PR98] and *MAXQ Value Function Decomposition (MAXQ)* [Die00]. Here the first and last one are introduced.

2.2.2.1. Option

Sutton [SPS99] firstly structured previous approaches in the learning with SMDPs and proposed the first comprehensive framework, *Option*, to solve the reinforcement learning problem in the SMDP environment.

The macro-actions here are called *options* $o \in \mathcal{O}$. Each option has three components $\langle I, \pi, \beta \rangle$, where

- $I \subseteq \mathcal{S}$ is the initiation set which specifies if this option is available at the current state,
- $\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$ is the above mentioned policy for this macro-action,
- $\beta : \mathcal{S} \to [0,1]$ is the probability that the macro-action terminates at the current state.

Note that for primitive actions these three components are defined as $\pi(a|s) = 1$, $\beta(s) = 1$ and $I = S$.

Suppose that an option o is invoked at time t and terminates after τ steps. The reward of this option is defined as the discounted cumulative reward within this period, as Eq. 2.10 describes.

$$r^o = \sum_{k=1}^{\tau} \gamma^{k-1} r_{t+k}(s, s', a) \tag{2.10}$$

The state transition probability from state s to s' is written as Eq. 2.11

$$p^o_{ss'} = \sum_{k=1}^{\infty} p(s', k|s, o)\gamma^k, \tag{2.11}$$

where the $p(s', k|s, o)$ represents the probability of option o terminating at state s'. The policy over options is defined as $\mu : \mathcal{S} \times \mathcal{O} \to [0,1]$. Referring to Eq.2.8, the value functions in the *option* framework are written as Eq. 2.12:

$$\begin{aligned} V^\mu(s) &= \sum_{o \in \mathcal{O}} \mu(s, o)[r^o + \gamma^\tau V^\mu(S_{t+\tau})] \\ Q^\mu(s, o) &= r^o + \sum_{s'} p^o_{ss'} \sum_{o'} \mu(o'|s') Q^\mu(s', o') \end{aligned} \tag{2.12}$$

The Q-learning [WD92] was firstly adopted to learn the policy [SPS99] based on the value functions in SMDP. But the update rule in traditional Q-learning does nothing until the current option terminates. This sometimes restricts the algorithm's performance due to the delayed update. The interruption of an option was explored to address such problem: when another option is better evaluated under current state, the current option policy is interrupted and the better one is invoked. However, this requires the evaluation of all options at every state. Later, [Pre00] proposed the intra-option learning method to achieve flexible options and proved their convergence to iterative optimality.

2.2.2.2. MAXQ Value Function Decomposition

Dieterrich [Die00] developed the *MAXQ* framework to solve the hierarchical RL problem. Instead of having one single SMDP, a hierarchy of multiple SMDPs (called subtasks M_i) are built based on the decomposition of the original MDP which is generalized as the root task M_0 in the hierarchy of SMDPs. The solutions to all subtasks can be learned simultaneously. To solve the root task M_0, a series of its subtasks M_i are executed. Recursively, to solve the subtask M_i, the sub-subtasks are executed according to the policy π_i of M_i until primitive task is encountered.

This hierarchical structure is shown in Fig. 2.3 as a task graph with the example in the taxi domain. There are a total of four places R, G, B, Y in the environment. The passenger appears randomly at one of four places and has a destination of another one. The taxi driver needs to navigate to the passenger, pick up the passenger, navigate to the destination and drop off the passenger.

A hierarchical task graph is manually designed to structure this problem into a hierarchical reinforcement learning problem. The scenario, i.e., take the passenger to the destination, is abstracted as the root task M_0. Under *root* task there exist two subtasks M_i: *Get* and *Put* which are called the children of M_0. The subtask *Get* is further decomposed into *Pickup*, a primitive task, and *Navigate*, a sub-subtask that is again decomposed into four primitive tasks.

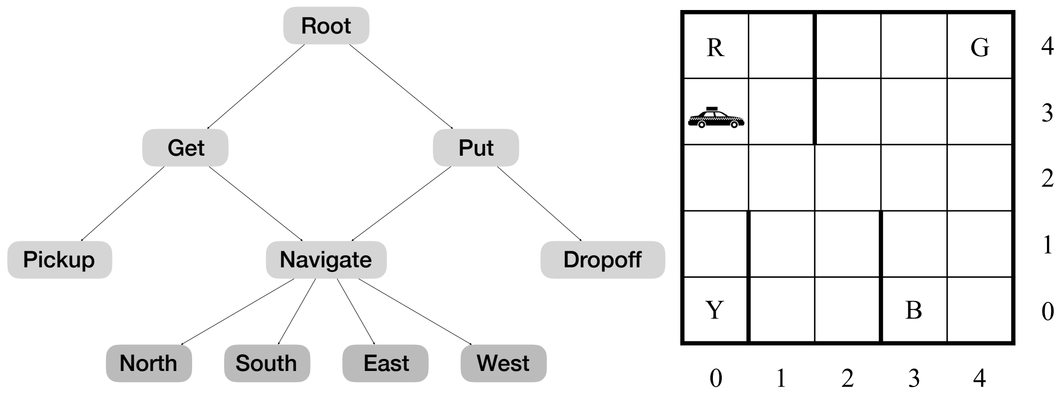


Figure 2.3.: R, G, B, Y are four locations in this 5×5 grid world with bold lines representing obstacles. The passenger appears randomly in one of these four places and has a random destination in the rest three locations. The taxi moves one cell each time and needs to bring the passenger to destination.

The (sub)task is similar to an *option* with the three components $\langle I, \pi, \beta \rangle$ in Sec.2.2.2.1. The differences are twofold as illustrated in [BM03]. On the one hand, the (sub)tasks in the MAXQ framework explicitly own a set of lower tasks (in other words, child tasks). On the other hand, the MAXQ framework adopts the pushdown stack mechanism in the implementation. The newly chosen subtask is pushed to the top of stack and will be firstly executed at each time step. After execution, the terminal conditions of all remaining tasks in the stack are examined and the terminated tasks are popped. The solution to this hierarchical task graph is defined as a hierarchy of policies $\pi = \{\pi_0, \ldots, \pi_n\}$ with π_i being the policy of subtask M_i. The value function of the task M_i with policy π_i, denoted as $V^{\pi_i}(s)$, is written as

$$V^{\pi_i}(s) = E[\sum_{k=0}^{\infty} \gamma^k r(s, s', a)|s_0, \pi_i] \tag{2.13}$$

Assuming that within subtask M_i the subtask M_j is chosen and takes N steps to finish according to its policy π_j, the state-action value function for choosing M_j is written as Eq. 2.14

$$Q^{\pi_i}(s, M_j) = E[\sum_{k=0}^{N-1} \gamma^k r(s, s', a)|s_0, \pi_j] + E[\sum_{k=N}^{\infty} \gamma^k r(s, s', a)|s_N, \pi_i] \tag{2.14}$$

Obviously, the first part is the value function of task M_j at state s_0, in other words, the discounted cumulative reward for executing the child task M_j. Eq. 2.14 can be further rewritten as the Bellman equation form

$$Q^{\pi_i}(s, M_j) = V^{\pi_j}(s) + \sum_{s', N} p^{\pi_j}(s', N|s, j)\gamma^N Q^{\pi_i}(s', \pi(s')) \tag{2.15}$$

The latter part is called *completion function* and Eq. 2.15 is re-written as Eq. 2.16.

$$Q^{\pi_i}(s, M_j) = V^{\pi_j}(s) + C^{\pi_i}(s') \tag{2.16}$$

The completion term C can be further decomposed according to Eq. 2.15 until primitive tasks are encountered. As a result, the value function of the *root* task can be recursively decomposed as the sum of the value functions of sub-tasks. This decomposition can be implemented in the tree search algorithms, where a nested tree is built for each task and the value function can be easily calculated (see Sec. 2.3.2).

2.2.3. Monte Carlo Tree Search

Monte Carlo Tree Search can be viewed as a variant of simulation-based search methods, as illustrated in Sec. 2.2.1, which builds a forward search tree and uses the Monte Carlo simulation to evaluate the states (or nodes) in this tree. A thorough overview of MCTS and its extensions is given in [BPW+12]. One iteration in MCTS consists of four steps: selection, expansion, simulation and backpropagation that are visualized in Fig. 2.4 with the example of two driving cars.

Each node stores the information describing the current state variables, the visit count N and $Q(s, a)$ of all available actions. Selection of actions is represented by the edges and the resulted state by executing the selected action is in the direct child node. One iteration starts from the root node and keeps selecting promising child node until a not fully expanded node is reached (selection). A new node is then expanded (expansion) and the simulation starts from the expanded node until the termination conditions are met (simulation). Finally the information gathered along this path is backpropagated to all visited nodes in this iteration so that the state-action values $Q(s, a)$ are updated (backpropagation). After that a new iteration starts and the selection uses the newest backpropagated tree to select a child node until the leaf node. After enough iterations the algorithm converges to an optimal action sequence. We usually limit the number of iterations considering the computational budget. Several strategies for the final selection

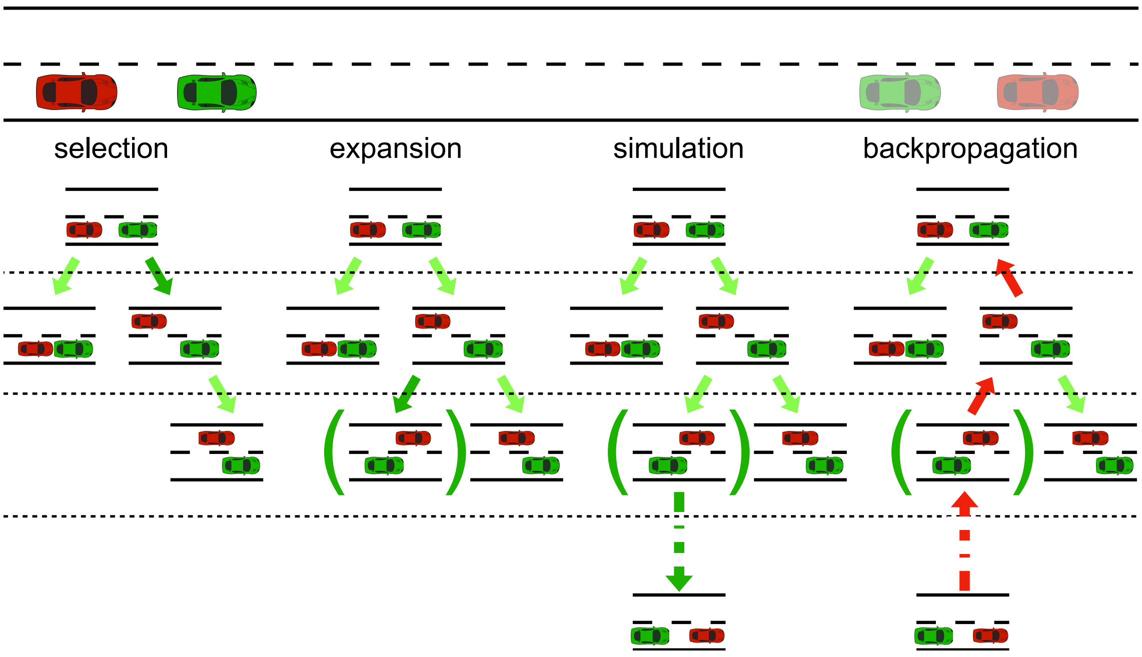


Figure 2.4.: Monte Carlo Tree Search in the example of an overtaking maneuver: the selection phase descends the tree by selecting promising children until a node is encountered that has not yet been fully expanded. Upon expansion random actions are simulated until the terminal conditions are fulfilled. The result is backpropagated through all nodes along the chosen path. Eventually the algorithm converges to an optimal action sequence.

in the built tree are available, such as choosing the action which has the highest action value (*max* child), the highest visit counts (*robust* child) or the highest Upper Confidence bound Tree (UCT) value (*UCT* child).

MCTS has many variants and the most commonly used is the UCT, which treats the selection as a multi-arm bandit problem and uses the Upper Confidence Bound (UCB1) [ACBF02] in Eq. 2.17 to guide the selection process. The selection chooses the child that maximizes

$$UCB1 = Q(s,a) + C_p\sqrt{\frac{2\ln N(s)}{N(s,a)}}. \tag{2.17}$$

The first term $Q(s,a)$ focuses on the exploitation and the second term stresses the exploration with $N(s)$ being the visit count of the parent node storing the state s and $N(s,a)$ being the visit count of the child node, i.e., the successor of the parent node taking action a. The constant C_p is used to balance the exploration and exploitation dilemma: the larger the C_p is, the stronger the exploration is. It is suggested by [KSW06] that C_p equals $\sqrt{2}$ with $Q(s,a) \in [0,1]$.

2.2.4. Monte Carlo Tree Search in Simultaneous Move Games

The success of MCTS in turn-based games has been proven in [SHM$^+$16b, SSS$^+$17b]. Simultaneous move games, in which all players make decision at the same time, is not a trivial extension for MCTS, because the unknown state of other players requires that MCTS should be able to handle the imperfect information. A simultaneous move game can be visualized as a *matrix game* in Fig. 2.5, which depicts a 2-agent scenario with an action space of 2 primitive actions. The number of possible outcomes in this scenario is $2^2 = 4$ and totally 5 nodes are shown with one parent node and 4 child nodes.

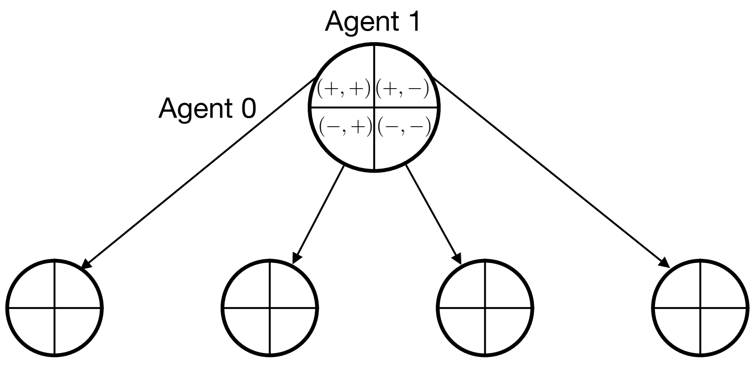


Figure 2.5.: Matrix Game with 2 agents owning 2 primitive actions each: acceleration + and deceleration −. Four child nodes with different joint actions belong to the root node. The root node is fully expanded.

During the selection phase of MCTS, each agent makes its own decision without knowing others decisions. The individually chosen actions $a^{(i)}$ form the joint action $\boldsymbol{a}$ and $\boldsymbol{a}$ determines the transition to the child node. Several selection strategies are proposed, such as the *Decoupled UCT* [BF09], *Exp3* [ACBFS95], *Sequantial UCT* [DTW11], etc. Here the method *Decoupled UCT* is introduced.

In DUCT, each agent i keeps recording its own action values $Q^{(i)}(s,a)$ and visit counts $N^{(i)}(s,a)$ at state s, which means that $Q^{(i)}(s,a)$ is the average value from all simulations that contain $a^{(i)}$ and $N^{(i)}(s,a)$ is the sum of visit counts of all child nodes which contain $a^{(i)}$. Recall the example in Fig. 2.5, the $N(s)$ is thus 4 and $N^{(0)}(s,+)$, $N^{(0)}(s,-)$, $N^{(1)}(s,+)$, $N^{(1)}(s,-)$ are all equal 2. The UCT calculation takes place individually in each agent and is thus decoupled, as Eq. 2.18 shows. Consequently, a new node is expanded when the joint action is not encountered before, which is different from the classical MCTS which expands node as long as it is not fully expanded. Theoretically, the tree can grow deeper at the third iteration if these two agents made different choices (+ and −) in the first two iterations.

$$DUCT = Q^{(i)}(s,a) + C_p\sqrt{\frac{2\ln N(s)}{N^{(i)}(s,a)}} \tag{2.18}$$

2.3. Related Work

This section introduces the related work in the fields of cooperative automated driving and planning with macro-actions.

2.3.1. Cooperative Automated Driving

Instead of treating other traffic participants as unchangeable obstacles, cooperative planning considers other's anticipation and reaction of the ego vehicle's behavior and tries to attain the global optimum. The first successful demonstration of cooperative automated vehicles emerged from the California PATH program [SH96] in the 1990s, where the notion of string stability is introduced to maintain the stability of a group of automated vehicles. Later in 2007, [SFK07] introduces the research project on cooperative vehicles in Germany which explores the potential of cooperative perception and action planning and outlines the software and hardware structures.

Till now there exists no standard definition about cooperative behaviors. In this work, we adopt the definitions from [DP14] which specifies cooperative behavior and its necessary preconditions, as Fig. 2.6 shows.

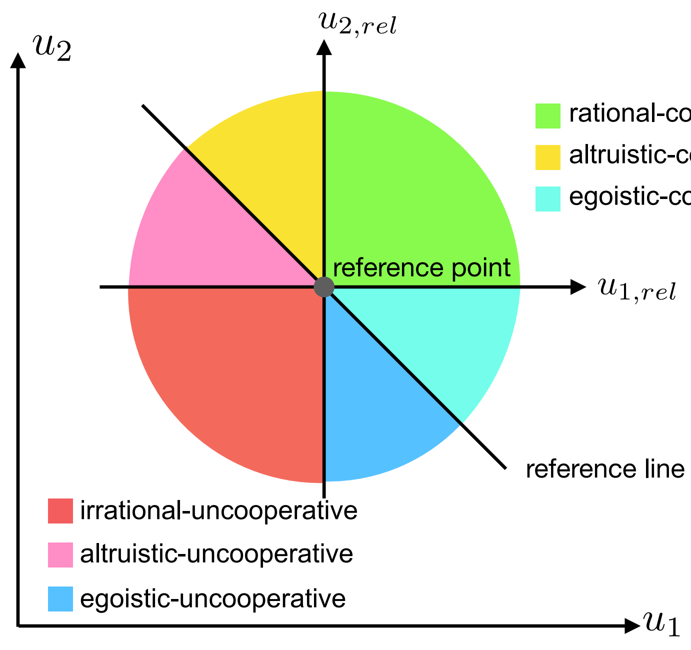


Figure 2.6.: Definition of Cooperative Behaviors [DP14]: the gray point is the reference point of current maneuver combination. The reference line passes through this point and has a slope of -1. All maneuver combinations which locate above this reference line are considered cooperative. The cooperative combinations can be further divided based on the relative gain/loss of each agent's utility.

Besides the uncooperative and cooperative behaviors, the cooperative behaviors are further divided into *altruistic*, *rational* and *egoistic* cooperation based on the increase and decrease on the own and others' utilities. A set of possible cooperative maneuvers are predefined based on quintic polynomial trajectories which are optimized considering the safety, energy, time, as well as driving comfort. Assuming that utilities of all agents can be perfectly estimated and that they are perfect substitutes, the algorithm based on the exhaustive search finds the maneuver combination with the maximum utility within the set of possible combinations. Since all agents are homogeneous, every agent will find the same maneuver combination in a decentralized manner. The algorithm is tested under two scenarios: one vehicle needs to merge in the other vehicle's current lane, both of two vehicles need to merge to the same lane. However, this method is not suitable for larger domains since the exhaustive search is computationally demanding and the heterogeneous environment factors, such as uncooperative vehicles, are not considered.

Similarly, cooperation in the merge scenario is detailed analyzed in [MLA17] based on the pre-defined maneuver templates. However, each template is not designed for a single vehicle but a group of maneuvers for all vehicles, thus requiring a centralized decision making. The templates are derived based on optimal control theory. Both lateral and longitudinal movements are optimized with physical constraints. During the online operation the feasibility of each template is checked according to the current situation using exhaustive search. Since the templates are designed for all agents, the exhaustive search has a computational complexity of $\mathcal{O}(n)$ with n being the number of templates.

[LKK16] first explored the potential of the Monte Carlo Tree Search algorithm for cooperative driving. They implemented Information-Set MCTS (ISMCTS) presented in [CPW12] to ensure decoupled decision making, and conducted decentralized planning. Similar to [DP14] they define a set of high-level actions, resembled by action primitives, with an action duration of 1 time step (1 second). They demonstrate the algorithms capabilities in three different merge scenarios, with up to three vehicles directly interacting with the ego vehicle, while the others were merely guided by an IDM. However, they are restricting the number of lane changes in the planning horizon to one.

[SSSS16] applies deep reinforcement learning methods for maneuver planning in a double merge scenario. They design an option graph to specify the routine of decision making. For example, a traverse on this graph can be *root* $\rightarrow$ *merge* $\rightarrow$ *left change* $\rightarrow$ *go* $\rightarrow$ *accelerate*, where the first four items can be regarded as macro-actions and *accelerate* is a primitive action. Each node in the graph finds its policy represented by a neural network after the offline training process. Experiments in light, mild and dense traffic demonstrate promising result.

To the author's best knowledge, current search on cooperative automated driving mostly requires a lot of offline work on the maneuvers, such as predefined maneuver template or offline trained neural network. Online generation of maneuver plans suffers the scalability problem and has a relatively shallow planning horizon.

2.3.2. Monte Carlo Tree Search with Macro-Actions

Till now, approaches that integrate the macro-actions into MCTS can be divided into two categories considering the learning process of macro-actions:

- pre-defined/offline learned macro-actions;
- online simultaneous learning of the policy over macro-actions and their respective execution policy;

[PWC12, PPW+14] adopt the first idea and define macro-actions as the repetition of actions for solving the physical traveling salesman problem. It is pointed out that different durations of macro-actions are difficult to realize because the agent always prefers the action with longer durations. Experiments show that these macro-actions yield much better results than MCTS with primitive actions. However this approach is too restricted and cannot be generalized to other applications considering their definitions of macro-actions.

[dWRB16] investigates MCTS with macro-actions in the application of computer games. They introduce Option Learning MCTS (OL-MCTS) which maintains the classical MCTS procedure but replaces the action space with defined options (macro-actions). The internal policies of macro-actions are pre-defined, offline trained or based on third algorithms. For example, the macro-action `go to location A` invokes an A^* planner. Experiments shows that OL-MCTS outperforms classical MCTS in 13 out of 18 games .

[PRHK17] uses a Deep Q-Network (DQN) [MKS+13] to train a set of macro-actions offline and integrate these learned macro-actions into the action space of MCTS, which achieves more flexible macro-actions to an extent than traditional pre-defined ones since each action is a trained neural network and has certain generalization ability. But this method requires an offline storage of the trained macro-actions. The performance of the resulting MCTS is strongly dependent on the training result of macro-actions.

The approaches based on the second idea are presented in [TT15, BSR16]. [TT15] adopts the MAXQ framework and proposes a hierarchical MCTS algorithm, where each macro-action is learned by a nested MCTS in the larger search tree and the simultaneous learning of a hierarchy of policies is realized. The evaluation in the Taxi-domain shows very promising result in terms of learning speed and optimality.

2.3.3. Macro-Actions in Multi-Agent Systems

[GMM+06, LSO+17, SSSS16, AKK14] provide very good example in implementing macro-actions into multi-agent systems. The following three challenges need to be addressed:

- **Asynchronous decision making**

  As illustrated in Sec. 2.1.3, there exist three schemes: t_{all}, t_{any} and $t_{continue}$. In a decentralized environment, macro-actions of different agents end asynchronously. Since the first two schemes t_{all} and t_{any} demands synchronization of decision making, $t_{continue}$ is the only one suitable for our case.

- **Flexible design of macro-actions**

  [SPS99] pointed out that naive pre-defined macro-actions are even more harmful than only planning with primitive actions. To mitigate the risk, the policy inside a macro-action should be learned online and be flexible according to the current situation.

- **Cooperation Level**

  [GMM+06] shows that learning of macro-actions can be distracted by lower level actions of other agents in the multi-agent-system and defines a cooperation level which restricts cooperation to the higher level. Another similar example is the modeling of localized macro-actions that are quite common in partially observable environments [LSO+17, AKK14, Omi15], where the agents do not consider cooperation at the level of primitive actions. Modeling of this kind is useful when localized macro-actions are sufficient, such as in the classical resource-collection, surveillance-and-rescue domains. However, for cooperative automated driving, cooperation is required for both macro-actions and primitive actions to solve conflict scenarios. Neither localized macro-actions nor restriction of cooperation of higher levels is applicable.

3. Underlying Conditions and Preliminaries

3.1. Underlying Conditions

Before introducing the concepts of our algorithm, the underlying conditions are explained as follows:

- The learning is based on self-play mechanism: The ego agent is assigned with index of 0 and has its own modeling of other agents, thus formulating a multi-agent system where the learning process takes place.
- The modeling of other agents refers to the estimated desire and reward function of others. The structure and parameters of the reward function among different agents can vary with agents' property, such as higher cost of deceleration and acceleration of trucks than personal cars.
- The ego agent can perfectly percept the state of other agents, i.e., the state is always fully observable.
- The state variables of an agent include the longitudinal and lateral positions (x and y), current lane, velocities ($\dot{x}$ and $\dot{y}$) and accelerations ($\ddot{x}$ and $\ddot{y}$).

3.2. Preliminaries

The problem of decentralized planning with macro-actions is formulated as a decentralized (semi-)MDP (see Sec. 2.1.3) which is solved using hierarchical reinforcement learning methods. The solution to this Dec-(S)MDP is the joint policy of all agents $\Pi = \{\pi^{(1)}, \pi^{(2)}, \dots\}$.

3.2.1. Nomenclature

We use superscript i, j with bracket $^{(i)}$, $^{(j)}$ to indicate that one variable is related to an agent. The subscript $_k$ indicates ordinary differences, such as different time steps. Besides, the upper and lower cases of one symbol usually (except N and n) have similar implication but the upper case refers to the variable itself while the lower case represents a specific value of this variable. Other symbols and their definitions are collected in Table 3.1

Table 3.1.: Nomenclature

symbol	**Implication**
t	time with unit second
s, S	state
a, A	action
π, μ	policy, the probability of choosing an action given the state
r, R	reward
γ	discount factor, representing the importance of successor states on the current state
N	number of an element, usually the visit count of a node or an action
n	node
g, G	return, the discounted cumulative reward
x, y	longitudinal and lateral position of the vehicle rear axle midpoint
l	index of lanes starting from 0
$V^{\pi}(s)$	state value at state s, the expected return under policy π
$Q^{\pi}(s,a)$	state-action value for taking action a at state s under policy π
t_{sim}	step length of one planning cycle, also the duration of the primitive actions
w_s	weight of the longitudinal acceleration in the reward function
w_d	weight of the lane change in the reward function
w_v	weight of the velocity deviation to the desired velocity in the reward function
w_l	weight of the lane deviation to the desired lane in the reward function
$r_{collision}$	penalty when the vehicle collides
$r_{invalid}$	penalty when the vehicle drives off the road
γ	discount factor when calculating the potential based shaping term and the return
ϵ	balance between the greedy and random selection
C_p	weight of the exploration term in the UCT calculation

4. Concepts

4.1. Planning with Primitive Motions

We firstly conduct the introduction of classical MCTS for cooperative action planning and use it as a baseline approach. Many parts in this section can be directly adopted when planning with macro-actions in Sec. 4.2: the definition of primitive actions serve as a basis of macro-actions; Moreover, the basic ideas of monte carlo learning of action values and decentralization in planning with primitive actions also hold in planning with macro-actions.

4.1.1. Action Space

Intuitively, the action space contains only primitive actions, which means that one action lasts only one time step. Totally five actions are defined: *acceleration*, *deceleration*, *do nothing*, *left change* and *right change*. Each action is described with the changes in longitudinal velocity $\Delta\dot{x}$ and lateral position Δy. The position of a vehicle refers to the lateral and longitudinal position of the midpoint of rear axle in the world coordinate. The lateral and longitudinal movements are calculated based on the fifth-order polynomials in Eq. 4.1 with Δy, $\Delta\dot{x}$ and other boundary conditions. For detailed derivation please refer to [WKZG12] and [DP14]. The solved trajectories of these five primitive actions are depicted in Fig. 4.1.1.

$$\begin{aligned} x(t) &= a_5 \cdot t^5 + a_4 \cdot t^4 + a_3 \cdot t^3 + a_2 \cdot t^2 + a_1 \cdot t + a_0 \\ y(t) &= b_5 \cdot t^5 + b_4 \cdot t^4 + b_3 \cdot t^3 + b_2 \cdot t^2 + b_1 \cdot t + b_0 \end{aligned} \tag{4.1}$$

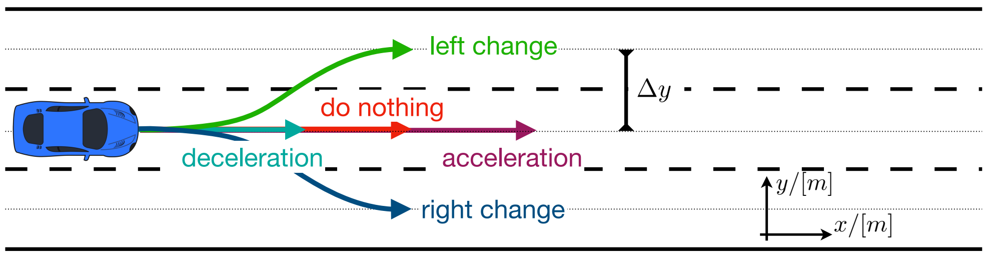


Figure 4.1.: Trajectories of the five primitive actions

4.1.2. Reward Function

The problem of determining the reward for each agent in a multi-agent system is studied in the area of *Credit Assignment*. In the multi-agent system, a reward is given by the environment when a joint action is executed. The simplest solution is to divide this reward evenly between all agents, which is called *global reward*. Clearly, the lazy agent and hard-working agent are thus equally rewarded. Under some circumstances, the agent cannot receive sufficient feedback to its specific action, which might lead to poor scalability in difficult problem [WT02]. Besides, when the global reward is not accessible, such as the distributed computation application like the robot foraging domain [Win09], the credit assignment must be done in other ways.

As opposed to the global reward, the other extreme is the *local reward* which is solely a feedback of the agent's own action regardless of others' behaviors. This approach encourages individual contribution but would result in a selfish learned policy. [Mat94] introduced *social reinforcement* to the individual reward. Two types of social reinforcement are defined: *observational reinforcement* obtained by observing others' behaviors, and *vicarious reinforcement* obtained whenever other agents are directly rewarded. The final reward function is a weighted sum of the above mentioned components and the experiment in the robot foraging domain shows better performance than the local reward.

In our application of cooperative automated driving, the problem is even more complicated: there exists actually no explicitly practical goal for each agent – solving a scenario with conflicting interests while maximizing the overall reward given each vehicle's safety, efficiency and comfort preferences. Here we design the reward function similar to the social reinforcement and calculate the reward in two parts:

4.1.2.1. Local Reward

The local reward, i.e., the feedback of each agent's own action includes three parts: action cost, collision/invalidity penalty and potential based reward.

Action Cost

The action cost is calculated based on the longitudinal acceleration and the number of changed lanes according to [AKM17], as Eq. 4.2 shows.

$$r_{action} = w_s \int \ddot{x}^2 dt + w_d \Delta l, \tag{4.2}$$

where Δl indicates the number of changed lanes and $w_s, w_d < 0$ mean that the agent is always trying to keep constant velocity and lateral position. Notice that the five primitive actions are pre-defined with constant parameters and the derived trajectories are also constant, so the action cost stays constant and a more complex function for the action cost has equivalent effect with the tuning of the parameters w_s, w_d.

Collision and Invalidity Penalty

Besides the action cost, we also define a very large negative value to penalize actions which lead to collision with other vehicles $r_{collision}$ and off-road $r_{invalid}$. Considering the existence of observational reinforcement (illustrated later), the collision penalty would double if the cooperation factor $\lambda = 1$, which should be considered during the tuning of these two parameters.

Potential Based Reward Shaping

In practice, the desire of each agent also needs to be considered as a reward term. For our work the desire is expressed as as a certain velocity and lane position. Potential Based

Reward Shaping (PBRS) is used here to model the reward for getting closer to the desired state, i.e., if the taken action brings the agent closer to its desired state, it will be rewarded positively. As [NHR99] pointed out, PBRS can accelerate the convergence of the learning process and is optimality invariant.

A potential function $\phi(s)$ is defined to determine the potential of each state. This function 4.3 is only dependent on the agent itself and has the global maximum at the desired state with monotonically increasing from left and decreasing to the right side.

$$\phi(s) = \Phi - w_v|v_c - v_d| - w_l|l_c - l_d| \tag{4.3}$$

In Eq. 4.3, v_c, l_c represent current velocity, current lane respectively and v_d, l_d are the desired ones. Φ is a constant dependent on the deviation between initial and desired states. In practice, it can be calculated at the beginning of the planning cycle based on the deviation of initial state and desired state, as Eq. 4.4 shows.

$$\Phi = w_v|v_{ini} - v_d| + w_l|l_{ini} - l_d| \tag{4.4}$$

w_v and w_l describe the importance of the velocity deviation and lane deviation with v_{ini} and l_{ini} being the initial velocity and lane. Based on such formulation, the potential function has a global optimum of Φ at the desired state and equals 0 at the initial state. The closer the agent is to the desired state, the higher the potential value will be. The potential based for transition from state s to s' is written as

$$r_\phi = \gamma\phi(s') - \phi(s), \tag{4.5}$$

where $\gamma \in [0, 1]$ is the discount factor representing importance of future reward (see sec. 2.1.1). Essentially, r_ϕ describes the change in the state. Thus, the local reward function for each agent can be written as:

$$r^{(i)}_{local} = r^{(i)}_{action} + r^{(i)}_{collision} + r^{(i)}_{invalid} + r^{(i)}_{\phi} \tag{4.6}$$

4.1.2.2. Observational Reinforcement

Assuming that each agent has perfect perception of others actions, such as velocity, acceleration and all agents share the same parameters w_s and w_d, the action cost of other agents can be assumed. The potential based shaping term could be also assumed if the desired state of other agents are known or can be estimated. Thus the observational reinforcement is calculated the same as local reward using Eq. 4.6. Similar to [DP14, LKK16], the cooperation factor λ (from $\lambda^{(i)} = 0$ *egoistic*, to $\lambda^{(i)} = 1$ *fully cooperative*) is introduced as the weight of this term, as Eq. 4.7 shows.

$$r^{(i)}_{ob} = \lambda^{(i)} \sum_{j=0, j\neq i}^{n} r^{(j)}_{local} \tag{4.7}$$

The resulting reward function is shown in Eq. 4.8 and can be called the cooperative reward.

$$r^{(i)} = r^{(i)}_{local} + \lambda^{(i)} \sum_{j=0, j\neq i}^{n} r^{(j)}_{local}, \tag{4.8}$$

where $r_{local}^{(i)}$ refers to the local reward and the latter is the observational reinforcement, more specifically, the discounted sum of assumed reward of other agents $r_{local}^{(j)}$ based on perception.

4.1.3. Monte Carlo Learning

Based on the cooperative reward function, the return in one iteration starting from t with length τ is calculated with $G^{(i)} = \sum_{k=0}^{\tau} \gamma^k r_{t+k}^{(i)}$. Note that the return $G^{(i)}$ is for the joint action $\boldsymbol{a}$ from the standpoint of agent i instead of the its own action $a^{(i)}$, because the reward $r^{(i)}$ that each agent receives is based on the joint action $\boldsymbol{a}$. Clearly, when all agents have the same cooperation factor $\lambda^{(i)} = 1$, the return $G^{(i)}$ in one iteration is the same for each agent.

As explained before in Sec. 2.2.1, $Q(s, a)$ ca be approximated without bias by the average of returns from all iterations. By replacing the a with $\boldsymbol{a}$ in Eq. 2.9 and applying it in the MCTS, the joint action value w.r.t. agent i , i.e., $Q^{\pi^{(i)}}(s, \boldsymbol{a})$ can be written as Eq. 4.9

$$Q^{\pi^{(i)}}(s, \boldsymbol{a}) = \frac{1}{N^{(i)}(s, \boldsymbol{a})} \sum_{n=1}^{N} G_n^{(i)}(s, \boldsymbol{a}), \tag{4.9}$$

where $G_n^{(i)}(s, \boldsymbol{a}) = \sum_{k=0}^{\infty} \gamma^k r^{(i)}$ is the return in the n-th iteration. Note that this Q-value is for the joint action $\boldsymbol{a}$ in the view of agent i instead of the single action $a^{(i)}$ because the reward $r^{(i)}$ that each agent receives is based on the joint action $\boldsymbol{a}$. Clearly, when cooperation factors of all agents $\lambda^{(i)}$ equal 1, they have the same $Q^{\pi_i}(s, \boldsymbol{a})$.

4.1.4. Decentralization

Since agents are not communicating, the action planning needs to be conducted in a decentralize manner, where each agent can only control its own action rather than the joint actions for all agents. As explained in Sec. 2.2.4, the agent's action-value estimation $Q(s, a)$ cannot distinguish among all joint actions containing this agent's action and the Decoupled-UCT should be used in the MCTS. Thus the agent i needs to estimate the state-action-value $Q(s, a^{(i)})$ based on all $Q(s, \boldsymbol{a})$ that have $a^{(i)}$ at i-th position of $\boldsymbol{a}$. Techniques in distributed reinforcement learning in [LR00] are adopted to address the decentralization problem. We conduct the marginalization among all $Q^{\pi^{(i)}}(s, \boldsymbol{a})$ to get $Q^{\pi^{(i)}}(s, a)$ with Eq. 4.10 based on the calculation of *DUCT* Eq. 2.18.

$$Q^{\pi^{(i)}}(s, a^{(i)}) = \frac{1}{N(s, a^{(i)})} \sum_{\boldsymbol{a}} 1(\boldsymbol{a}[i] = a^{(i)}) N(s, \boldsymbol{a}) Q^{\pi^{(i)}}(s, \boldsymbol{a}), \tag{4.10}$$

where $N(s, a^{(i)}) = \sum_{\boldsymbol{a}} 1(\boldsymbol{a}[i] = a^{(i)}) N(s, \boldsymbol{a})$ and $1(\boldsymbol{a}[i] = a^{(i)})$ returns 1 when the i-th position of $\boldsymbol{a}$ is $a^{(i)}$) and 0 otherwise.

Combining Eq. 4.9 and 4.10, we get

$$Q^{\pi^{(i)}}(s, a^{(i)}) = \frac{1}{N(s, a^{(i)})} \sum_{\boldsymbol{a}} 1(\boldsymbol{a}[i] = a^{(i)}) G^{(i)}(s, \boldsymbol{a}) \tag{4.11}$$

Eq. 4.11 means that the state-action value $Q^{\pi^{(i)}}(s, a^{(i)})$ is the mean of all returns after executing the joint action $\boldsymbol{a}$ that contain $a^{(i)}$. This average can be written in an incremental form Eq. 4.12 which is used as the update rule in the backpropagation step.

$$Q^{\pi^{(i)}}(s, a^{(i)}) \leftarrow Q^{\pi^{(i)}}(s, a^{(i)}) + \alpha[G^{(i)}(s, \boldsymbol{a}) - Q^{\pi^{(i)}}(s, a^{(i)})] \tag{4.12}$$

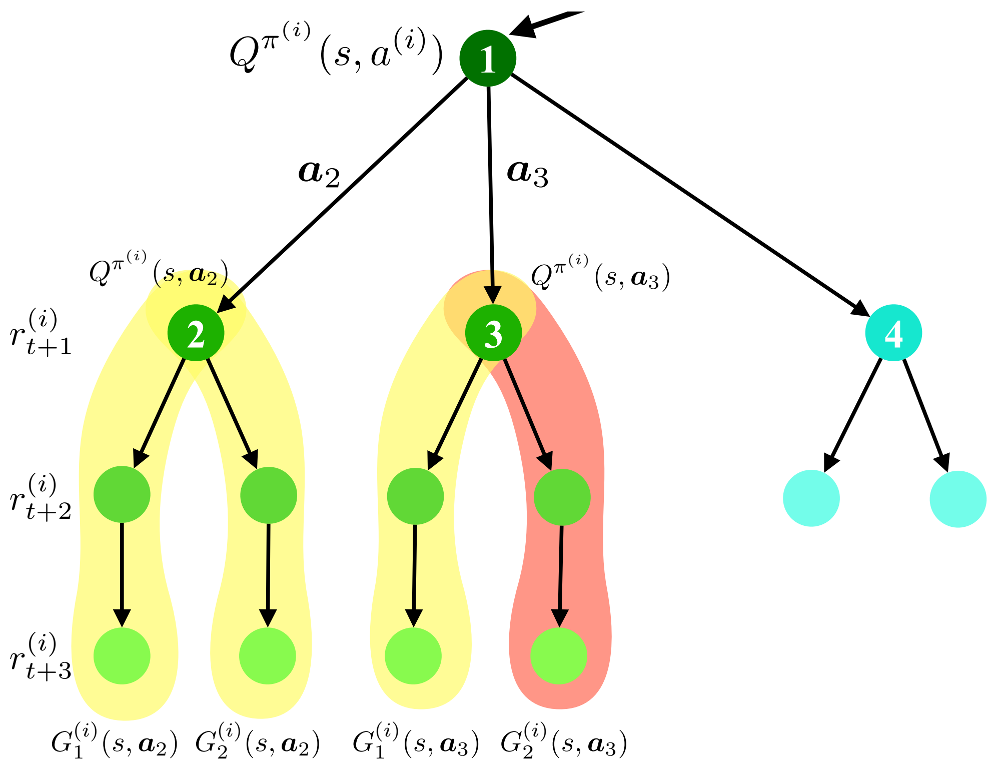

$Q^{\pi^{(i)}}(s, a^{(i)})$
1
$\boldsymbol{a}_2$
$\boldsymbol{a}_3$
$Q^{\pi^{(i)}}(s, \boldsymbol{a}_2)$
$Q^{\pi^{(i)}}(s, \boldsymbol{a}_3)$
$r^{(i)}_{t+1}$
2
3
4
$r^{(i)}_{t+2}$
$r^{(i)}_{t+3}$
$G^{(i)}_1(s, \boldsymbol{a}_2)$
$G^{(i)}_2(s, \boldsymbol{a}_2)$
$G^{(i)}_1(s, \boldsymbol{a}_3)$
$G^{(i)}_2(s, \boldsymbol{a}_3)$

Figure 4.2.: Update rule with example of a subtree starting from node 1

where $\alpha = \frac{1}{N(s,a)}$. This derivation process is visualized in Fig. 4.2 with a subtree starting from node 1 followed by three child nodes $2, 3, 4$. The red shadowed path is the newest iteration and its information will be backpropagated to node 1. For agent i the joint actions which lead to node 2 and 3, i.e., $\vec{a}_2$ and $\vec{a}_3$ have action a at position i. Eq. 4.9 is represented by $Q^{\pi^{(i)}}(s, \boldsymbol{a}_2) = \frac{1}{2}(G^{(i)}_1(s, \boldsymbol{a}_2) + G^{(i)}_2(s, \boldsymbol{a}_2))$. The old $Q^{\pi^{(i)}}(s, a)$ is calculated based on the three yellow shaded iteration and can be written as $Q^{\pi^{(i)}}(s, a) = \frac{1}{3}(G^{(i)}_1(s, \boldsymbol{a}_2) + G^{(i)}_2(s, \boldsymbol{a}_2) + G^{(i)}_1(s, \boldsymbol{a}_3))$, representing the Eq. 4.11. The backpropagation at node 1 is written as $Q^{\pi^{(i)}}(s, a) \leftarrow Q^{\pi^{(i)}}(s, a) + \frac{1}{4}(G^{(i)}_2(s, \boldsymbol{a}_3) - Q^{\pi^{(i)}}(s, a))$ according to the update rule Eq. 4.12.

4.2. Planning with Macro-Actions

As explained in Chapter 1, temporal abstraction is considered to increase the effective search depth of MCTS by extending actions over several time steps (macro-actions). The structure inside the macro-actions are not fixed like the examples in Sec. 2.3.2. Instead, we adopt the approaches in the hierarchical reinforcement learning, more specifically, the *option* and *MAXQ* framework, integrate them into the Simultaneous Move MCTS so that all agents can simultaneously learn the policy over macro-actions and within macro-actions. We name this approach *Decentralized Hierarchical Monte Carlo Tree Search*. The following sections explain how the three challenges mentioned in Sec. 2.3.3 are addressed.

4.2.1. Design of Macro-Actions

Considering the listed conflict scenarios that require cooperative driving in [UGA+15], we propose four macro-actions: *overtake*, *merge in*, *make room* and *to desired velocity*. The decomposition techniques from the *MAXQ* framework is used to construct the hierarchical action graph as Fig. 4.3 shows. Based on the definitions in the *Option* framework, each

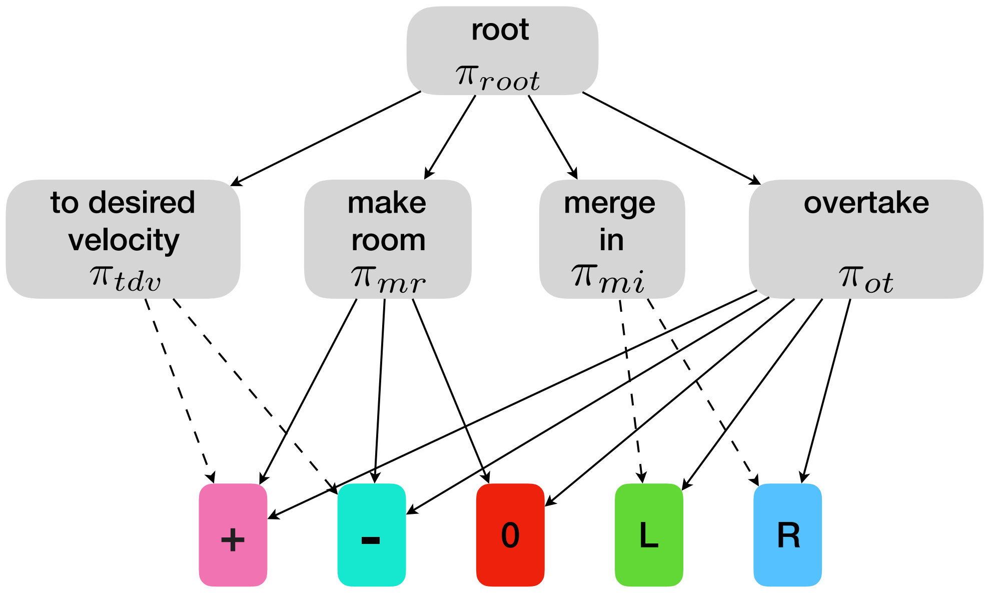


Figure 4.3.: Hierarchical Action Graph

macro-action has four components $\langle I, \pi, \mathcal{A}, \beta \rangle$, where $\mathcal{A}$ is the set of available actions at the immediate lower level. The primitive actions all come from Sec. 4.1, i.e., *acceleration*(+), *deceleration* (-), *do nothing* (0), *lane change left* (L) and *lane change right* (R). Each macro-action has a possible subset of these primitive actions. Note that *to desired velocity* and *merge in* are connected with their $\mathcal{A}$ with a dotted line, which means that the primitive actions are not always available at the current state. For example, when the current velocity is smaller than the desire, only *acceleration* is available for the *to desired velocity*. The solution to the whole scenario is generalized as the *root* macro-action that entails all lower macro-actions. The initiation set I (or initial condition) and termination probability β (or termination condition) of all actions are defined per Table 4.2.1.

Macro-Action	Initial Condition	Terminal Condition
overtake	behind slower vehicle (w.r.t. ego vehicle) and left lane exists	in front of slower vehicle
merge in	not in desired lane	in desired lane
make room	always possible	always possible
to desired velocity	not at desired velocity	at desired velocity

Table 4.1.: Initial and terminal conditions for macro-actions

Each macro-action has its own policy, thus forming a hierarchy of policies for each agent i: $\pi^{(i)} = \{\pi_{root}, \pi_{tdv}, \pi_{mr}, \pi_{mi}, \pi_{ot}\}$, where π_{root} is namely the policy over macro-actions and others are policies within macro-actions. The learning process takes place in all levels and converges simultaneously. [BSR16] shows that prior knowledge can accelerate the learning process without harming the optimality compared with a random initial policy. This may not hold true when the state space is extremely large, as [SSS+17a] presented in the Go-game that without human knowledge the program can achieve super-human performance. In order to achieve faster convergence, we also implement the simple prior knowledge for the macro-action *overtake* according to the common sense of executing an overtake maneuver. Greedy in the Limit with Infinite Exploration (GLIE) technique is a technique to balance the prior knowledge and the random exploration (see Sec. 5.2.1.2). The comparison of the convergence between two versions is conducted in Sec. 6.3.2. and shows that prior knowledge can accelerate the learning and is optimality invariant.

It should be mentioned that the hierarchical action graph also applies in the planning of primitive actions, where only the macro-action *root* exists with five child actions, namely the five primitive actions. Thus we can simply change the action graph to switch between two versions of planning.

4.2.2. Termination Scheme

There are three termination schemes mentioned in Sec. 2.3.3: t_{all}, t_{any} and $t_{continue}$. Obviously, t_{all} and t_{any} which require centralized decision making are not suitable for our application. We use the $t_{continue}$ scheme during the learning process, which means that the macro-action of a certain agent terminates according to the state without being influenced by others.

4.2.3. Reward Function

In SMDPs the reward for a macro-action is defined as the discounted cumulative rewards within this macro-action according to Eq. 2.10 [SPS99]. Note that this definition does not consider the potential based shaping term that we mentioned in Sec. 4.1.2. Considering the definition of the potential-based reward shaping, the shaping term for a macro-action m is written as Eq. 4.13 with τ being the duration of this macro-action.

$$r_{m,\phi} = \gamma^{\tau}\phi(s_{t+\tau}) - \phi(s_t) \tag{4.13}$$

As a result, the shaped local reward for a macro-action m of agent i in the multi-agent system is

$$r^{(i)}_{m,local} = \sum_{k=1}^{\tau} \gamma^{k-1} r^{(i)}_{t+k,action} + \gamma^{\tau}\phi^{(i)}(s_{t+\tau}) - \phi^{(i)}(s_t) \tag{4.14}$$

It can be proven that Eq. 4.14 is equivalent to the discounted sum of the shaped local rewards in each single step, as Eq. 4.15 shows.

$$r^{(i)}_{m,local} = \sum_{k=1}^{\tau} \gamma^{k-1} r^{(i)}_{t+k,local} \tag{4.15}$$

Eq. 4.15 also applies in the multi-agent system, which simply replaces the local reward $r^{(i)}_{local}$ with the global reward $r^{(i)}$, as Eq. 4.16 shows.

$$r^{(i)}_{m} = \sum_{k=1}^{\tau} \gamma^{k-1} r^{(i)}_{t+k} \tag{4.16}$$

4.2.4. Learning with Hierarchically Bounded Return

The learning process of this hierarchy of policies is based on the MAXQ framework. As mentioned in Sec. 2.1.1, the value function is defined based on the policy. Equations 2.13 and 2.14 point out that discounted cumulative reward is constrained within the current following policy. (Note that the policy in the classical MDP settings has no specified termination condition, so the ∞ is written in the definition of value function Eq. 2.4.) Applying this notion to our problem we get the definition of *Hierarchically Bounded Return*:

Lemma: The return of choosing child action a_c at state s under parent action a_p is the discounted cumulative rewards starting from s, following π_c then π_p of the parent action until π_p ends.

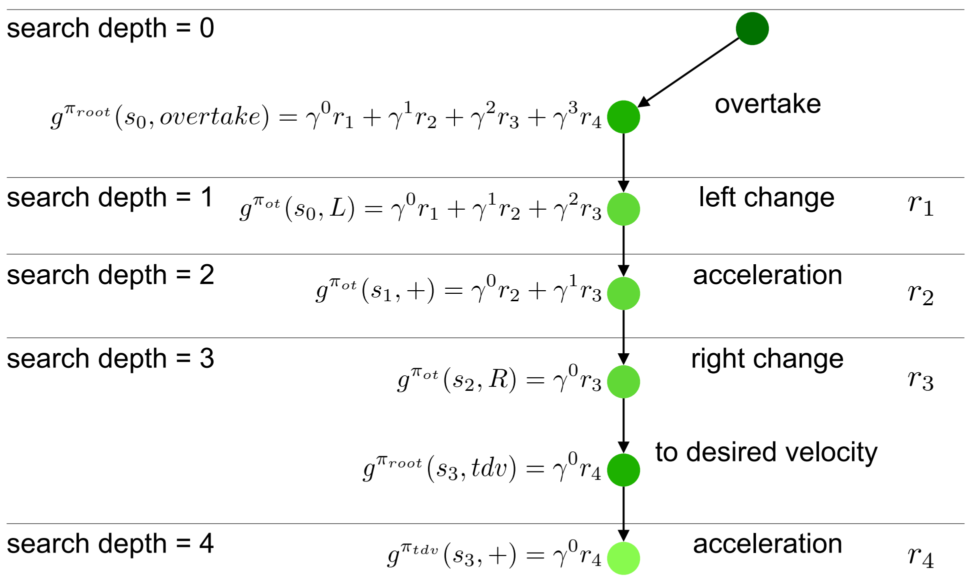

search depth = 0
$g^{\pi_{root}}(s_0, overtake) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \gamma^3 r_4$
overtake
search depth = 1
$g^{\pi_{ot}}(s_0, L) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3$
left change
r_1
search depth = 2
$g^{\pi_{ot}}(s_1, +) = \gamma^0 r_2 + \gamma^1 r_3$
acceleration
r_2
search depth = 3
$g^{\pi_{ot}}(s_2, R) = \gamma^0 r_3$
right change
r_3
$g^{\pi_{root}}(s_3, tdv) = \gamma^0 r_4$
to desired velocity
search depth = 4
$g^{\pi_{tdv}}(s_3, +) = \gamma^0 r_4$
acceleration
r_4

Figure 4.4.: Hierarchically Bounded Return in the example of single agent

Suppose there exists one iteration with search depth 4 in the single agent domain as Fig. 4.4 shows. The nodes where the decisions on the macro-action are made are also explicitly shown. Note that the macro-action alone is not executable and thus cannot transfer the state, that's to say, state transitions are only possible when primitive actions are decided. The rewards for all primitive actions are listed on the right side and the calculation of returns on the left. It can be seen that the return for choosing macro-action *overtake* under policy π_{root} includes rewards from r_1 to r_4 since π_{root} terminates at $depth = 4$. Besides, $V^{\pi}(a, s)$ and $C^{\pi}(i, s, a)$ in Eq. 2.16 are respectively represented by $r_{overtake} = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3$ according to Eq. 4.16 and $\gamma^3 r_4$. Comparatively, the return for choosing *left change* under policy $\pi_{overtake}$ only includes the reward from r_1 to r_3 because *overtake* terminates at $depth = 3$.

The notion of hierarchically bounded return also holds in the multi-agent system, where the reward for a single action is replaced by the cooperative reward according to Eq. 4.8 and 4.16. Considering the $t_{continue}$ termination scheme, the boundary of return is only dependent on the own agent's execution of macro-actions. Similar to the single agent system, the state transition and reward are only possible when all agents make a decision on the level of primitive actions. Hence, some agents need to make several decisions in one time step from the *root* level until they reach the level of actions, e.g., $root \rightarrow overtake \rightarrow acceleration$, while others only need to make one decision if they are already within a macro-action, e.g., $make\ room \rightarrow deceleration$.

Based on the hierarchically bounded return, the action value $Q^{\pi_i}(s, a)$ is calculated according to Eq. 4.11 and the update rule is the same as Eq. 4.12.

With the update rules described in the above sections 4.1 and 4.2, the state-action values $Q(s, a)$ are learned with multiple iterations based on MCTS. It can be seen as a table which specifies value of each action in each state and we can find the optimal policy by maximizing over all $Q(s, a)$. Note that for planning with macro-actions, the output is a decision sequence, such as $root \rightarrow overtake \rightarrow acceleration \rightarrow \ldots$. The agent only execute the first primitive action and then starts a new planning cycle. Whether the previously learned macro-actions are preserved in the new cycle is optional. Planning a new cycle from scratch is called *hierarchical* mode and preserving the information from last cycle is called *polling* mode. Sec. 5.5 provides more details about this topic.

5. Implementation

Based on the concepts of planning with primitive actions and macro-actions, the implementation is illustrated in this chapter. Considering that the planning with primitive actions serves as a baseline approach and can be seen as a simplification of the planning with macro-actions (see Sec. 4.2.1), the following sections focus on the implementation of MCTS with macro-actions.

5.1. Overview

The algorithm named *Decentralized Hierarchical Monte Carlo Tree Search*, abbreviated as DecH-MCTS, is outlined in Algorithm 1. All parameters and action space are defined offline. In every planning cycle, MCTS builds a search tree of possible future states starting from the root node (capturing the initial state) within a given computational budget, usually in form of CPU-time, number of iterations, etc. Policies over and within the macro-actions are learned when building the search tree. In each iteration four steps are executed: selection, expansion, simulation and backpropagation, where the selection and expansion steps are included in the tree policy 1. Both the tree and simulation policies are forward search and this forward search stops when it reaches the maximal search depth or a terminal state or the vehicles are collided or off-road. Then the backpropagation update all nodes along this path with newly gathered information.

Considering the notion of self-play mentioned in Chapter 3, each node stores a vector of agents containing the ego agent and other modeled agents. The node can be indexed by the search depth and joint action that leads to this node. Each agent has its action space (hierarchical action graph) and state space, including the velocities, accelerations and positions in both longitudinal and lateral directions, and the reward function.

5.2. Tree Policy

The tree policy includes the selection and expansion steps. The selection always prefers untried actions, which is followed by the expansion of a new node. If all actions have been already expanded, the selection chooses the action with the highest UCT value. Consequently, a node in MCTS with a single agent keeps expanding until all available actions have been tried and only after full expansion can the search tree grow deeper. By contrast, an agent in the decentralized setting (or simultaneous move games) independently selects the action with highest DUCT value and the choices of all agents form the joint

Algorithm 1: Decentralized Hierarchical MCTS

```
Function Planning($\Upsilon$: agent space, $\mathcal{A}$: action space, $\mathcal{R}$: reward sequence):
    $\boldsymbol{a} \leftarrow \emptyset$, $\mathcal{R} \leftarrow \emptyset$
    while problem not solved do
        new root node $n_0 \leftarrow n\langle \boldsymbol{a}, \Upsilon, \mathcal{A} \rangle$
        $a \leftarrow$ DecHMCTS($n_0, \mathcal{R}$)
        ExecuteAction($a$)
    return

Function DecHMCTS($n_0$: root node, $\mathcal{R}$: reward sequence):
    while computational budget not reached do
        $\langle n_{leaf}, \mathcal{R} \rangle \leftarrow$ TreePolicy($n_0$)
        $\mathcal{R} \leftarrow$ SimulationPolicy($n_{leaf}, \mathcal{R}$)
        BackPropagation($n_{leaf}, \mathcal{R}$)
    return $a \leftarrow$FinalSelection($n_0$)

Function TreePolicy($n$: node):
    do
        for $i \leftarrow 1$ to $|\Upsilon|$ do
            $a^{(i)} \leftarrow \epsilon - UCT(n, i)$
            $\boldsymbol{a} \leftarrow [\boldsymbol{a}, a^{(i)}]$
        $n \leftarrow$LookUp($n, \boldsymbol{a}$)
        if $n$ is not subnode then
            $\mathcal{R} \leftarrow$CollectReward($n, \boldsymbol{a}$)
    while $n \neq \emptyset$
    $\langle n_{leaf}, \mathcal{R} \rangle \leftarrow$ Expansion($n, \boldsymbol{a}, \mathcal{R}$)
    return $n_{leaf}, \mathcal{R}$

Function Expansion($n, \boldsymbol{a}, \mathcal{R}$):
    construct new node $n_{leaf}$ based on $n$
    if $\boldsymbol{a}$ is executable then
        Execution($n_{leaf}$, $\boldsymbol{a}$)
        $\mathcal{R} \leftarrow$CollectReward($n_{leaf}$, $\boldsymbol{a}$)
    return $n_{leaf}, \mathcal{R}$

Function SimulationPolicy($n$: node, $\mathcal{R}$: reward sequence):
    for $i \leftarrow 1$ to $|\Upsilon|$ do
        $a^{(i)} \leftarrow$ RandomSelection($\mathcal{A}^{(i)}$)
        $\boldsymbol{a} \leftarrow [\boldsymbol{a}, a^{(i)}]$
    if $\boldsymbol{a}$ is executable then
        Execution($n$, $\boldsymbol{a}$)
        $\mathcal{R} \leftarrow$CollectReward($n$, $\boldsymbol{a}$)
    return $\mathcal{R}$

Function BackPropagation($n$:node, $\mathcal{R}$: reward sequence):
    while $n \neq n_0$ do
        $N(n) \leftarrow N(n) + 1$
        for $i \leftarrow 1$ to $|\Upsilon|$ do
            $N(a^{(i)}) \leftarrow N(a^{(i)}) + 1$
            $G^{(i)} \leftarrow \sum^{a_p^{(i)}} \gamma^t r^{(i)}$
            $Q(a^{(i)}) \leftarrow Q(a^{(i)}) + \frac{G^{(i)} - Q(a^{(i)})}{N(a^{(i)})}$
        $n \leftarrow n_{parent}$
    return
```

action. If the formed joint action was not encountered before, the expansion will be executed. As a result, the node can grow deeper once each agent has tried its available actions once, which is independent of the number of joint actions.

5.2.1. Selection

The selection for the multi-agent system is decentralized. As a result, each agent must keep track of the visit count and action value of every available actions for the calculation of DUCT. In our implementation with C++, the `std::map` with keys being the action is adopted and values being the state-action values $Q(s,a)$.

5.2.1.1. UCT calculation

Considering that the exploration term $C_p\sqrt{\frac{2\ln N(s)}{N(s,a^{(i)})}}$ ususally lies around 1, the exploitation term $Q^{(i)}(s,a)$ is normalized according to the maximum and minimum among all available actions. Thus Eq. 2.18 is modified as follows

$$DUCT = \frac{Q^{(i)}(s,a) - \min_{a\in\mathcal{A}} Q(s,a)}{\max_{a\in\mathcal{A}} Q(s,a) - \min_{a\in\mathcal{A}} Q(s,a)} + C_p\sqrt{\frac{2\ln N(s)}{N(s,a^{(i)})}} \tag{5.1}$$

5.2.1.2. GLIE in the Selection

Selection according to the DUCT values is essentially a deterministic policy. We found that the number of expanded nodes of the root node is mostly limited to the number of available actions of each agent when we set the cooperation factor λ of all agents to 1, even if the number of iterations reaches up to 10,000. This problem is firstly discussed in [SSS09] which argues that the deterministic DUCT policy does not necessarily converge to a Nash equilibrium and can be exploited. This argument also holds in our situation:

Suppose that there are totally n agents with m available actions each. At the first m iterations, each agent has tried all available actions and formed m joint actions. In each iteration, all agents' rewards $r^{(i)}$ are the same because of $\lambda = 1$. Based on the update rule 4.12, the action value is equal to the r_i of the corresponding $\boldsymbol{a}$. At the $(m+1)$-th iteration, each agent chooses an action using the DUCT value based on the previous calculated Q and visit counts, i.e., $N(s) = m$ and $N(s,a) = 1$ for all child nodes. As a result, the selected joint action will be the one which was already selected in the last m iterations and has the largest r_i. The selection will adhere to this joint action until the exploration term dominates the difference in the DUCT calculation.

To address this problem, [SSS09] proposed to use an ϵ-Nash Equilibrium strategy where ϵ indicates how far we are from the equilibrium. The Counter Factual Regret (CFR in [ZJBP08], equivalent to the UCT), an ϵ-Nash Equilibrium strategy, is compared with the Decoupled UCT algorithm.

On the other hand, such problem can be also addressed by a stochastic DUCT policy. [GS07] has tried to introduce the idea of GLIE into the UCT policy to ensure the exploration when combining the online and offline knowledge. GLIE is a group of different techniques which injects limited randomness to the optimal policy 2.7 to ensure the exploration won't be harmed by the poor trials at the start. We adopt this idea in our implementation to provide the DUCT policy with stochasticity. The ϵ-Greedy, one of the GLIE techniques, is adopted and the adapted DUCT is named as $\epsilon - DUCT$. The probability of choosing action a, i.e., $\pi(a|s)$ is written as:

$$\pi^{\epsilon}(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}|} & \text{if } a = arg\,max_{a\in\mathcal{A}} DUCT(a) \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}, \tag{5.2}$$

where $\mathcal{A}$ is the set of available actions of size $|\mathcal{A}|$ and $\epsilon \in [0,1]$ controls the randomness. Larger ϵ means greater randomness. ϵ can be predefined or decay with the number of visit counts, i.e. $\epsilon = \frac{1}{N(s)}$. Obviously, the randomness by a decaying ϵ decreases as the visit count increases.

The $\epsilon - DUCT$ technique can be implemented in the following way:

Algorithm 2: $\epsilon - DUCT$

```
$\epsilon \leftarrow$ constant $\in [0,1]$ or $\frac{1}{N(s)}$
generate a random number $\xi \in [0,1]$
if $\xi \in [0,\epsilon)$ then
    select one action according to the DUCT values
else
    select one action randomly from available actions
return the selected action
```

5.2.2. Expansion

As stated before, when the formed joint action was not encountered, the expansion follows. Because we adopt the $t_{continue}$ scheme and different macro-actions have different duration, the selected joint action for the to-be-expanded node could contain a macro-action which is not executable. Only the joint actions which consist of primitive actions can trigger system transition and receive immediate reward. As a result, two kinds of implementation are available: pack all selections from macro-action to the primitive action in one node or explicitly expand the tree for every selection. The former requires complexer data structure for storing the action values and visit counts and needs complex backpropagation design. The data size of one node would be much larger. The latter one needs to construct nodes for the non-executable joint actions, which seems like a waste of nodes but realizes a much clearer data structure.

We adopt the latter approach and name the nodes for non-executable joint actions subnodes and introduce a new variable `searchDepth` to distinguish from the `treeDepth`. Obviously, `treeDepth` increments always at newly expanded nodes while `searchDepth` only does at non-subnodes.

In subnodes, the agent which has already chosen the primitive action needs to *wait* for others. It should be stressed that such *wait* is actually not waiting for a certain time but only a notion in terms of building a new node. This agent's available action at the subnode is only the current primitive action itself and will be preserved until the child node is reached. Only when all agents select primitive actions, the joint action can be executed according to the following section.

5.2.2.1. Execution

The hierarchical action graph is executed as a stack with LIFO rule. For the purpose of clearance, each agent has two stacks: `entryHierarchy` and `exitHierarchy`. `entryHierarchy` is constructed based on the `exitHierarchy` of the parent node with the newly selected action being pushed back to the top. Obviously, the macro-action *root* locates always at the bottom. If the node is non-sub, the joint action will be executed by each agent and the termination condition of each action in the `entryHierarchy` will be examined. The `exitHierarchy` is based on its own `entryHierarchy` with the terminated actions being popped out. Besides, the reward is only possible after the execution. The received reward together with these two stacks will be stored in an extern vector named as `rewardSequence`, which is detailed illustrated in Sec. 5.4.1

When executing primitive actions, the trajectory is calculated according to Eq. 4.1. The collision checking is based [ZS10], where vehicles are approximated by a certain number of circles. For implementation details please refer to [ZS10].

5.3. Simulation Policy

Domain knowledge can be implemented into the simulation phase to achieve better performance, as [DU07] and [GS07] discussed. In our problem, the prior knowledge of macro-actions is discussed in Sec. 4.2.1. Similar to [GS07] when introducing the domain specific knowledge, we employ the ϵ-Greedy technique in Algorithm 2.

Note that the simulation is essentially the same as the selection and expansion but does not build any new nodes into the tree. In the implementation a new node `simulationNode` is constructed copying the expanded node and serves as a container to run the simulation. The joint action saved within this node is overwritten when the simulation policy selects new one. Agents in the `simulationNode` keep updating their `entryHierarchy` and `exitHierarchy` also by overwriting the old ones. If the joint action is executable, the execution (see Sec. 5.2.2.1) is conducted and the reward as well as the current stacks are saved into the `rewardSequence`.

5.4. Backpropagation

The backpropagation (or backup) takes place after the forward search finishes. The visit count is firstly incremented by 1 and then the action value is updated according to the update rule.

In the application with classical simple games [BPW+12] where no immediate reward exists, only the final result of the game (win or lose) is propagated, which can be called *outcome*-based update. This operation is inappropriate in larger domains requiring deeper search because of the overwhelmingly delayed reward for each step. Additionally, we also want to evaluate each primitive action considering the driving comfort, energy consumption and safety. Thus the discounted cumulative reward, or return in other words, is used in our case. More specifically, the hierarchically bounded return (see Sec. 4.2.4) is implemented to realize the simultaneous learning over and within the macro-actions.

5.4.1. Reward Collection

Because of the tree structure, each node has a single parent node but many child nodes. A pointer to the parent node is saved in the child node and the parent node has an arrary of pointers to its child nodes. Tracing the path back to the root node from the current leaf node is very simple but descending from the root node to a certain child node is much more complicated. This results in difficulties when calculating the hierarchically bounded return.

This value can be relatively easily realized in the single-agent domain or the multi-agent system with t_{any} or t_{all} termination schemes, where the synchronization of decisions on macro-actions is possible. For example, [TT15, BSR16] implement the MCTS in a recursive manner. For each invoked macro-action a nested MCTS is built. When the nested MCTS terminates, i.e., current (macro-)action terminates, the reward, reached search depth and state are returned so that the bounded return can be easily calculated.

In our case with the $t_{continue}$ scheme, the decisions are asynchronous so the tree search must be built in a plain manner, i.e., four steps including selection, expansion, simulation and backpropagation. However, this demands a boundary check on each reward after (including itself, if applicable) the current node to determine whether it is relevant for

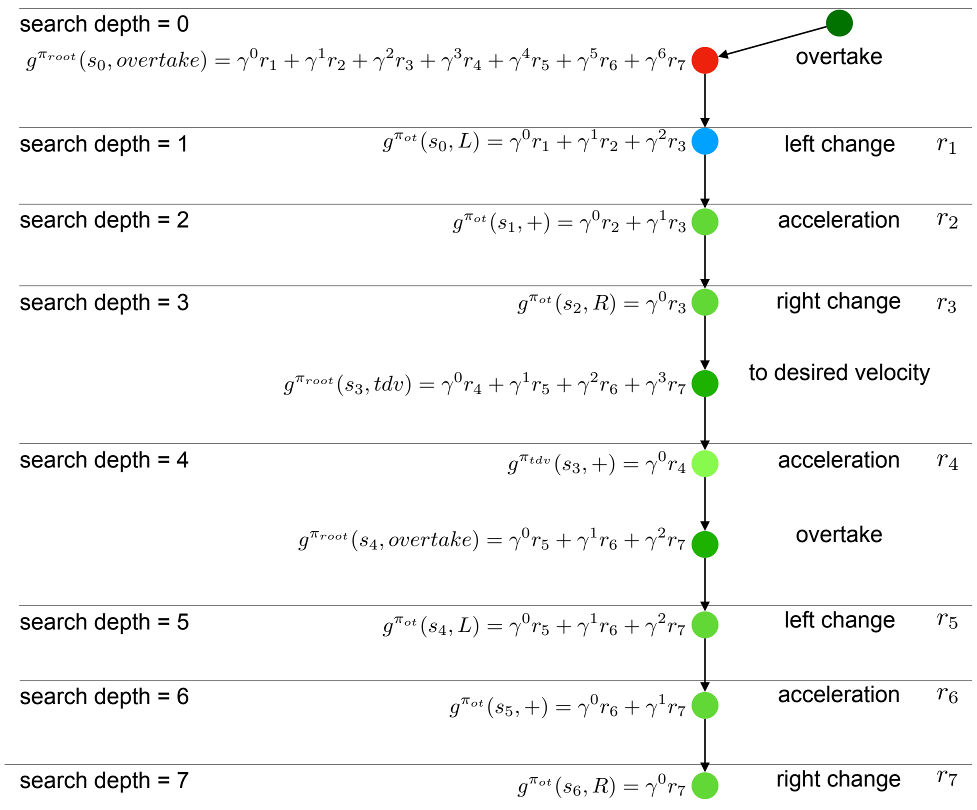

search depth = 0
$g^{\pi_{root}}(s_0, overtake) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \gamma^4 r_5 + \gamma^5 r_6 + \gamma^6 r_7$
overtake
search depth = 1
$g^{\pi_{ot}}(s_0, L) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3$
left change
r_1
search depth = 2
$g^{\pi_{ot}}(s_1, +) = \gamma^0 r_2 + \gamma^1 r_3$
acceleration
r_2
search depth = 3
$g^{\pi_{ot}}(s_2, R) = \gamma^0 r_3$
right change
r_3
$g^{\pi_{root}}(s_3, tdv) = \gamma^0 r_4 + \gamma^1 r_5 + \gamma^2 r_6 + \gamma^3 r_7$
to desired velocity
search depth = 4
$g^{\pi_{tdv}}(s_3, +) = \gamma^0 r_4$
acceleration
r_4
$g^{\pi_{root}}(s_4, overtake) = \gamma^0 r_5 + \gamma^1 r_6 + \gamma^2 r_7$
overtake
search depth = 5
$g^{\pi_{ot}}(s_4, L) = \gamma^0 r_5 + \gamma^1 r_6 + \gamma^2 r_7$
left change
r_5
search depth = 6
$g^{\pi_{ot}}(s_5, +) = \gamma^0 r_6 + \gamma^1 r_7$
acceleration
r_6
search depth = 7
$g^{\pi_{ot}}(s_6, R) = \gamma^0 r_7$
right change
r_7

Figure 5.1.: Hierarchically Bounded Return in an iteration of depth 7

calculation of the return. Clearly, the reward value alone or the action name is not enough for such operation. For such reason, the above mentioned `rewardSequence` is designed to store the reward values as well as the two stacks of actions: `entryHierarchy` and `exitHierarchy`. Together with the reward value, these three components are saved as an element in `rewardSequence`.

5.4.2. Computation of Hierarchically Bounded Return

Based on the collected *rewardSequence*, the boundary check between reward r and agent's action $a^{(i)}$ is conducted based on two criteria which are illustrated with the following example in the single agent domain:

Fig. 5.1 is an extended version based on the previous example Fig. 4.4. The search depth reaches up to 7 with three macro-actions being chosen at depth 0, 3 and 4.

- **Criterion 1**: The parent actions of the agent must exist in the parent actions of the reward.

  Suppose that we are conducting the boundary check between the reward r_4 and the agent's decision on choosing *overtake* at state s_0 (in red). The `entryHierarchy` of r_4 is *root*, *to desired velocity* and *acceleration* in order from bottom to top of the stack. The first two macro-actions are called as the parent actions of the primitive action *acceleration*. The agent's `entryHierarchy` consists of *root* which is the only parent action. Clearly this criterion is fulfilled. The implementation is realized by comparing the name of the each parent action.

- **Criterion 2**: The parent actions of the agent and parent actions of the reward must come from the same invoke.

Take the agent at search depth 1 (in blue) and the reward r_5 as an example. The parent actions of agent and reward are the same: *root* and *overtake*. However, the reward r_5 cannot be considered into the return of choosing *left change* under policy π_{ot} at state s_0 because the *overtake* of the agent is not the same one of the reward r_5. As a result, r_5 is not added to the calculation of $g^{\pi_{ot}}(s_0, L)$.

The implementation is realized by comparing the terminated actions staring from the node till this currently being checked reward. More specifically, another stack `terminatedHierarchy` representing the terminated (macro-)actions is firstly determined by substracting `exitHierarchy` from `entryHierarchy`. Then the `terminatedHierarchy` of each reward from this node to the target node (r_5) is examined to find if the parent action of the agent (*overtake* in this example) exists. If so, it means that the parent action has already terminated before reaching the target node and the reward r_5 from the target node is not relevant anymore.

These two criteria necessarily and sufficiently determine whether a future reward is relevant for the return of the current action. Besides, the exponent of the discout factor γ is determined by the difference of two search depths. For each subnode the difference should be reduced by 1 since no time elapsed at the subnode. The update of the action value is according to Eq. 4.12.

The above four sections depict how an iteration is done. These operations repeat many times to build an asymmetric forward search tree. This process can be terminated at any time as long as the predefined conditions are met. The following section shows how an action is chosen from the tree and executed.

5.5. Final Selection and Execution

Because we have built a tree in the decentralized way, the final selection should be also decentralized, which means that the formed joint action could be an untried one. However, such situation rarely occurs. It should be mentioned that the centralized final selection means that the agent selects the joint action which leads to the child node with highest action value $Q^{\pi^{(i)}}(s, \boldsymbol{a})$ or visit count $N(s, \boldsymbol{a})$, while the decentralized selection is based on the agents own track of the action value $Q^{\pi^{(i)}}(s, a)$ or visit count $N(s, a)$ (see Sec. 4.1.4). Obviously the centralized selection requires that other agents must choose the same joint action so that cooperation could be realized. [LR00] theoretically prove that the decentralized learning converges to the global optimum under the homogeneous agents settings as the centralized learning does. However, since the other agents do not always behave like the agent has assumed, the decentralized final selection is implemented for the sake of robustness.

As stated in Sec. 2.2.3 three possible selection policies are available: highest action value (*max* child), highest visit count (*robust* child) and highest UCT value (*UCT* child). In the evaluation we find that the former two usually produce the same results, which is also reasonable. The selection according to the highest visit count is also experimentally proven to have the best robustness in Sec. 6.3.3. The final selection based on *robust* child is described in Alg. 3. Due to the existence of macro-actions, the decentralized selection needs to be repeated until a primitive action is selected.

```
Algorithm 3: FinalSelection
Data: $n_0$: root node, $v$: ID of the ego agent
Result: $a^v$: action of ego agent
joint action $\boldsymbol{a} \leftarrow \emptyset$
do
    for $i \leftarrow 1$ to $|\Upsilon|$ do
        $a^{(i)} \leftarrow argmax_{a^{(i)}} N(s, a^{(i)})$
        $\boldsymbol{a} \leftarrow [\boldsymbol{a}, a^{(i)}]$
    $n \leftarrow$LookUp$(n, \boldsymbol{a})$
while $n \neq \emptyset$ and $\boldsymbol{a}$ is not executable
return $\boldsymbol{a}[v]$
```

It should be mentioned that the selected primitive action belongs to a certain macro-action. When starting a new tree search, such information can be maintained in the new tree search or discarded, which is respectively called *hierarchical* mode and *polling* mode (see Sec. 4.2). [SPS99] showed that the *polling mode* which starts a new planning cycle without any memory about the previous step provides better results because the polling mode allows the premature termination of macro-actions at each step and is thus more flexible. We adopt the polling control mode for the execution.

Moreover, the selection can be further executed and a sequence of planned actions can be extracted. The repeated selection terminates when the selected action is not reliable, for example, having insufficient visit count. We name this procedure the `ExtractActionSequence` and describe it as follows Alg. 4. The extracted plan can be used to examine the depth of effective search and how the learned macro-actions look like. The corresponding evaluation is conducted in Sec. 6.3.1

```
Algorithm 4: ExtractActionSequence
Data: $n_0$: root node; $C$: minimal required visit count
Result: $\vec{A}$: action sequence
action sequence $\vec{A} \leftarrow \emptyset$
joint action $\boldsymbol{a} \leftarrow \emptyset$
do
    for $i \leftarrow 1$ to $|\Upsilon|$ do
        $a^{(i)} \leftarrow argmax_{a^{(i)}} N(s, a^{(i)})$
        $\boldsymbol{a} \leftarrow [\boldsymbol{a}, a^{(i)}]$
    if $\boldsymbol{a}$ is executable then
        $\vec{A} \leftarrow [\vec{A}, \boldsymbol{a}]$
    $n \leftarrow$LookUp$(n, \boldsymbol{a})$
while $N(n) \geq C$
return $\vec{A}$
```

To sum up, this chapter shows the implementation details about the concepts in Chapter 4. The ϵ-greedy technique is introduced to ensure the exploration in the selection phase and combine the prior knowledge during the simulation. To calculate the hierarchically bounded return in the multi-agent system with $t_{continue}$ termination scheme of macro-actions, the rewards together with the corresponding stacks of actions are stored externally and the boundary check is conducted. After building the tree, the best action is selected in a similar decentralized manner and the polling control mode is adopted to execute the action.

6. Evaluation

In this chapter, the performance of the proposed DecH-MCTS algorithm is demonstrated based on the comparison with the classical MCTS. All test scenarios are firstly described, such as simple free drive, bottleneck, overtaking, etc. In each scenario two types of environments are available: homogeneous and heterogeneous. The homogeneous environment requires that all agents are equipped with the same planning algorithm with the same parameters. Consequently, there exists no difference between each agent's assumption about others. On the contrary, other agents in the heterogeneous environment can behave differently from what the ego agent assumes. The difference between assumption and reality is used to test the robustness of our algorithm.

We firstly demonstrate the general applicability of DecH-MCTS by finding a universal parameter setting. After that we conducted repeated experiments with different random seeds to compare the learning ability of DecH-MCTS and classical MCTS in terms of convergence speed and success rate. Lastly, the robustness of our proposed method is examined in a heterogeneous environment.

6.1. Test Scenarios

This section describe all used test scenarios for the evaluation of our algorithm. Note that the figures do not necessarily represent the precise coordinates of each vehicles but only relative positions are depicted. The initial states (position and velocity) are given in each specific experiment.

6.1.1. Free Drive

The free drive in Fig. 6.1 is the simplest scenario and is used to preliminarily validate the algorithm. Only one vehicle is driving on the middle lane of an endless straight road with a certain initial velocity. Different desired velocity and lane can be set to check if the algorithm behaves rationally.

6.1.2. Merge

The scenario *merge* is a typical conflict scenario which requires cooperation among the participating vehicles. We set this scenario on a three-lane straight road with the rightmost lane being blocked by a static vehicle shown in Fig. 6.2.

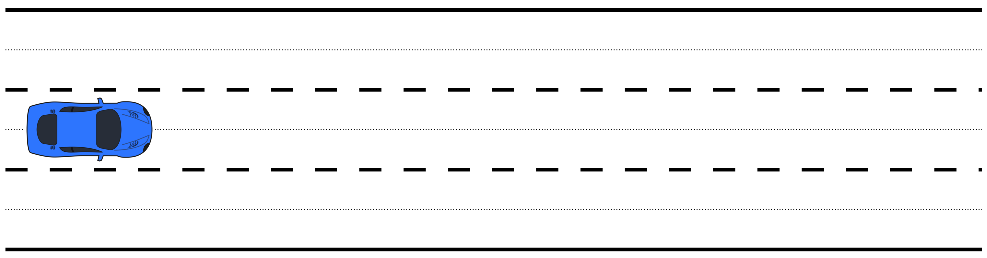


Figure 6.1.: Scenario: Free Drive with one vehicle

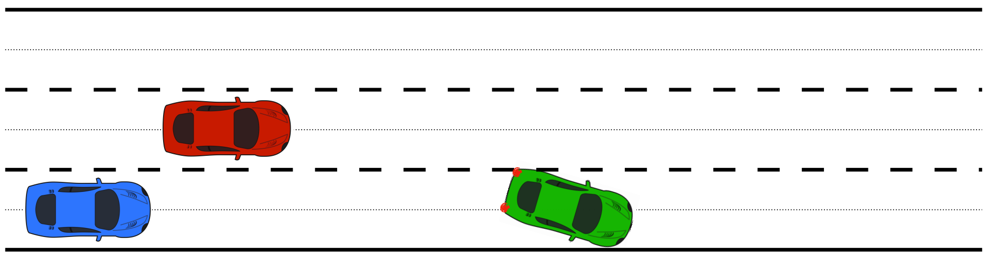


Figure 6.2.: Scenario: Merge; the green vehicle blocks one lane

6.1.3. Double Merge

The scenario *double merge* also requires cooperation among the participants, as Fig. 6.3 shows. The green vehicles block the leftmost and the rightmost lanes. Both the blue and red vehicles need to merge to the middle lane and pass the static vehicles.

6.1.4. Overtaking

The scenario *overtaking* is considered to validate the simultaneous convergence of the hierarchical policies in the DecH-MCTS algorithm and compare the performance between DecH-MCTS and the classical MCTS. Two variants are designed: overtaking between 2 vehicles and overtaking between 3 vehicles.

Fig. 6.4 depicts the simpler variant with 2 vehicles. The blue vehicle drives faster than the red one and needs to overtake. In the overtaking between 3 vehicles (Fig. 6.5), totally three vehicles are driving on the rightmost lane: vehicle 0 in blue has the highest velocity and needs to overtake both of the two front vehicles, vehicle 1 drives slower than vehicle 0 but faster than vehicle 2 (in red).

6.1.5. Bottleneck

The scenario *bottleneck* is another typical conflict scenario which usually happens in residential areas. As Fig. 6.6 shows, there are three participants on a narrow road with one lane in each direction. Vehicle 0 (blue) approaches from the left and vehicle 1 (red) comes from the right at different velocities. Vehicle 2 (green) blocks the lane of vehicle 0.

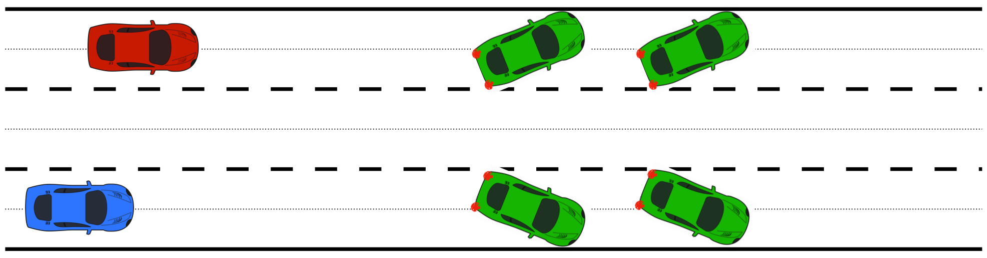


Figure 6.3.: Scenario: Double Merge; the green vehicles block two lanes

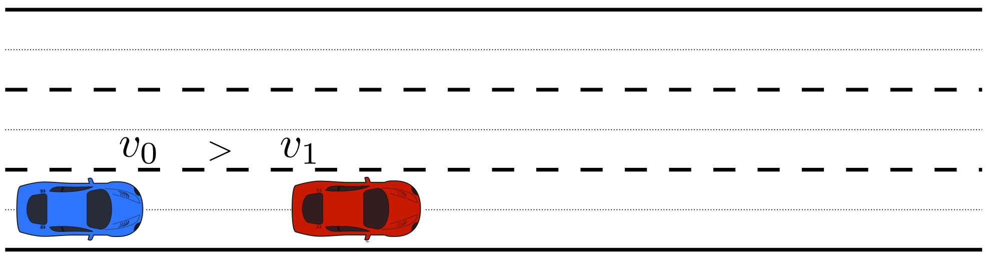


Figure 6.4.: Scenario: Overtaking between 2 vehicles

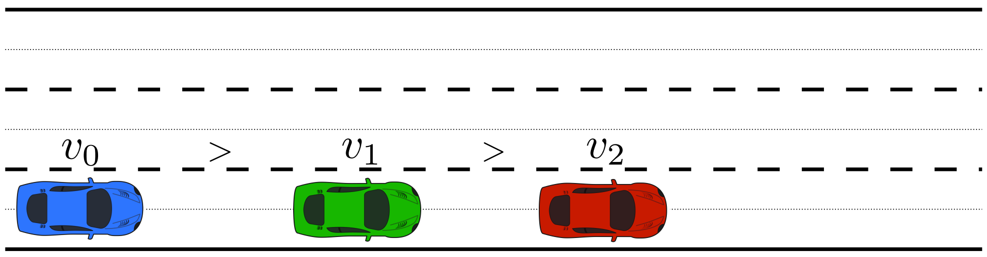


Figure 6.5.: Scenario: Overtaking between 3 vehicles

6.1.6. Open Loop

The scenario open loop is to test the performance under complex heterogeneous traffic. As explained before, only the ego agent in such environment is equipped with our planning algorithm, while others are controlled by SUMO, an open source traffic simulation developed by the DLR [KEBB12]. The vehicles in SUMO are set to move according to the IDM-Model. while the modeling in the algorithm of the ego agent can be different in terms of cooperation factor, reward functions, desired states, action space, etc. Fig. 6.18 shows an example situation during the testing.

6.2. General Applicability

We refer the general applicability to solving different scenarios with a universal parameter setting. Considering our original purpose of implementing the macro-action to achieve deeper search and accelerate the learning process, the number of iterations and search depth is the focus of our evaluation and can be varied with scenarios. All others should keep invariant in different scenarios, such as parameters in the reward function, UCT calculation, etc. Initial settings in each scenarios are listed in A.2.

Table A.1 in the appendix lists the tuning result. It should be mentioned that the optimality of these parameters is not guaranteed. On the one hand, finding the optimal parameter settings is impractical considering the amount of parameters and the varied scenarios. On the other hand, it is not necessary since our evaluation focuses on the flexibility, convergence and robustness. The parameter tuning is conducted to find the parameter values which make the algorithm work reasonably but not necessarily optimally.

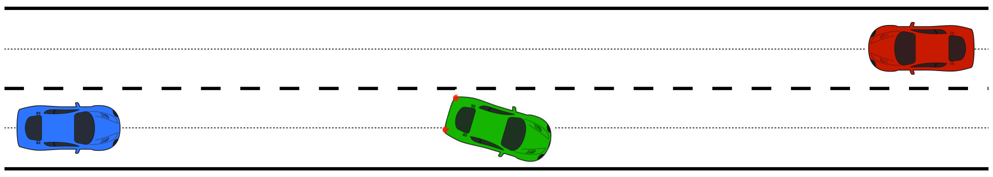


Figure 6.6.: Scenario: Bottleneck; the green vehicle blocks one lane

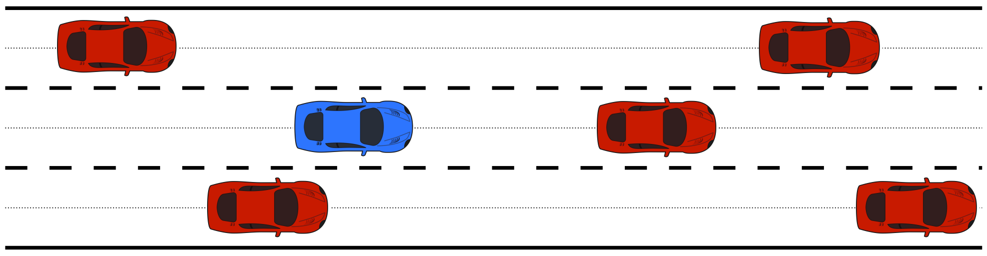


Figure 6.7.: Scenario: Open Loop with heterogeneous environment

Solutions of all scenarios are listed in A.3. It can be seen that our algorithm generates rational plans for all conflict scenarios as human would do. We choose the scenario *bottleneck* as an example for analysis, as Fig. 6.8 shows.

The upper graph shows 2D trajectories of both agents/vehicles. Color of points represents time according to the color bar on the right side. The black rectangle in the at $x = 1000\,\mathrm{m}$ on lane 0 is the static obstacle vehicle (agent 2). Between two black points is one planning cycle. We also visualize the whole plan sequence at each time step (see Sec. 5.5) with gray points. Changes of lateral and longitudinal velocity can be found in the lower four graphs. It can be seen that agent 0 firstly decelerates and then accelerates. These two agents meet at around $t = 8.2\,\mathrm{s}, x = 980\,\mathrm{m}$. At $t = 8s$ agent 0 changes to the left and then changes to right again to avoid collision with the obstacle.

6.3. Performance Analysis

Recall the original purposes of integrating macro-actions into MCTS, we propose the following three goals and uses different scenarios to conduct the experiment and analysis.

- flexible learning of macro-actions
- faster learning than the classical MCTS
- ability to generate robust solutions in the heterogeneous environment

6.3.1. Flexible Learning of Macro-Actions

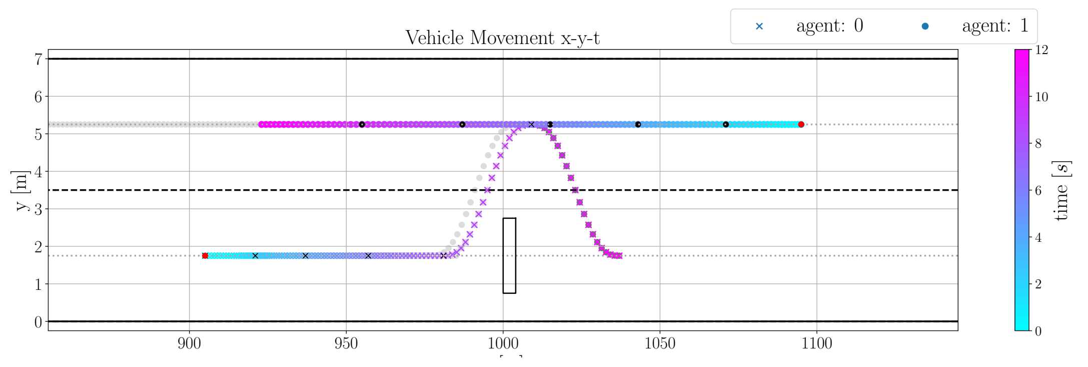
The scenario *overtaking* between 3 vehicles is considered to test the algorithm's ability to learn the execution of the macro-action *overtake* flexibly. The scenario is shown in Fig. 6.5 and the initial conditions are shown in Table 6.1.

Table 6.1.: Initial Condition for Scenario Overtaking between 3 vehicles

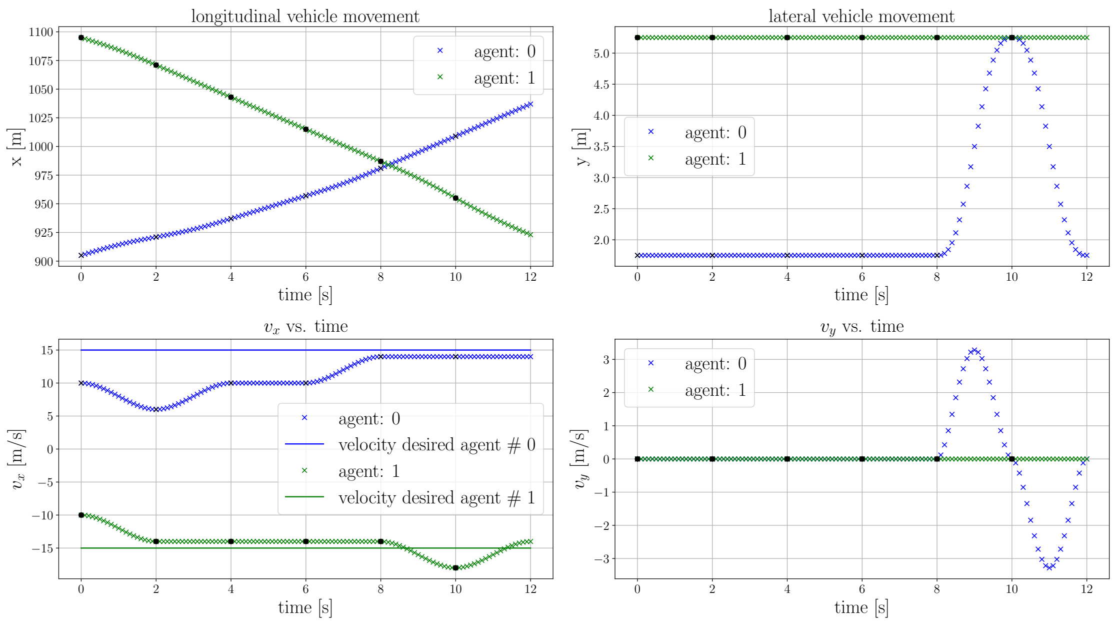
ID	**color**	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	blue	5	15	0	30	0
1	green	25	15	0	25	0
2	red	45	15	0	15	0

Agent 0 is defined as ego agent. The desired velocities and lanes of the other two agents are assumed to be known. All three vehicles are controlled by their own DecH-MCTS algorithm with the same parameter settings according to Table A.1 and $\lambda^{(i)} = 1$. The number of iterations is set to 2,000 and the maximal search depth to 20.

As mentioned in Sec. 5.5, the final selection can be repeated and a sequence of planned actions can be extracted. We extract the plan using algorithm 4 after the planning cycle



(a)2D Trajectory



(b)Details

Figure 6.8.: Solution for scenario: Bottleneck

Table 6.2.: Planned action sequence at $t = 0$

Depth	**Most Visited Action**		
	agent 0	agent 1	agent 2
0	overtake L	overtake L	make room 0
1	L	+	make room 0
2	+	+	make room 0
3	+	+	make room 0
4	+	R	make room 0

at the beginning. The terminal condition C is set as $\frac{1}{100}$ of total iterations, i.e., 20. The extracted action sequence is shown in 6.2.

To avoid ambiguity, the *wait* operations (see Sec. 5.2.2.1) of agent 0 and 1 at depth 1-4 are not shown. It can be seen that the effective search depth reaches to 4, indicating a plan for the future 10 s. Additionally, agent 0 learns the macro-action *overtake* differently from the agent 1: agent 0 needs to make two left changes till the leftmost lane and then accelerate to get in front of other 2 vehicles; what the agent 1 has learned about the execution of *overtake* contains only one lane change to the left, acceleration and then one lane change to the right.

Since all agents are modeled as homogeneous, the other two agents produce the same result. In the polling control mode, each agent executes the planned action, i.e., L, L, 0 respectively, and then starts a new plan without memorizing previous result. The state at $t = 2s$ is shown in Table 6.3

Table 6.3.: State after the first step (actions L, L, 0 are respectively executed)

ID	**color**	x [m]	v [m/s]	$lane$	v_d [m/s]	l_d
0	blue	35	15	1	30	0
1	green	55	15	1	25	0
2	red	75	15	0	15	0

The new planning cycle starting at $t = 2$ s produces the following result in Table 6.4. The *overtake* is newly learned at current state and the agents 0 and 1 have different learning results.

We run the algorithm repeatedly with the polling mode until the conflict scenario is solved. Fig. 6.9 shows the 2D trajectories of each agent, where the color of the data points represents the time according to the color bar on the right side. Between two black points is one planning step of 2 seconds. It can be seen that agent 1 changes to the left after driving in front of agent 2, while agent 0 stays at lane 2 until it gets in front of both two vehicles and then makes two lane changes to the desired lane 0.

Table 6.4.: Planned action sequence at $t = 2\,\mathrm{s}$

Depth	**Most Visited Action**		
	agent 0	agent 1	agent 2
0	overtake L	overtake +	make room 0
1	−	0	make room 0
2	+	0	make room 0

6.3.2. Faster Learning than Classical MCTS

The learning speed refers to the algorithm's performance at a certain number of iterations. Faster learning means lower requirement on the number of iterations to achieve the same performance.

Note that we do not use the computation time for evaluation because it varies with hardware. Appendix A.4 presents the variation of average computation time of one stage in different scenarios with different total iterations and maximal search depth (see Table 6.5). It can be seen that DecH-MCTS outperforms classical MCTS in terms of average computation time.

6.3.2.1. Performance Measure

Considering our application, three performance metrics are defined as follows:

- collision rate CR
- success rate SR
- undiscounted return G

Success rate specifies how often the conflict scenarios are solved within the given steps. Note that it does not equal $1 -$ *collision rate*. The third measure is the undiscounted sum of ego agent's reward at each step. We calculate the reward with the same parameters adopted in building the search tree.

We combine these three measures into one called *utility U* according to Eq. 6.1

$$U = \overline{G_{uncollided}} + CR \cdot r_{collision} + SR \cdot r_{success}, \tag{6.1}$$

where $\overline{G_{uncollided}}$ is the mean of undiscounted return of uncollided experiments. With the reward parameter settings based on A.1, $\overline{G_{uncollided}}$ lies usually between -20 and 20. $r_{collision}$ is not necessarily the same as that in the reward function but should have larger absolute value than $\overline{G_{uncollided}}$. Both $r_{collision}$ and $r_{success}$ are used to combine the above mentioned three measures and distinguish the successful/collided solutions from the short-sighted ones (see Sec. 6.3.2.3).

6.3.2.2. Independent Variables

There are three independent variables: scenarios, number of iterations and maximal search depth. The scenarios should be long enough, in other words, require enough many steps to solve the conflicts. Note that the maximal search depth refers to the possible maximal

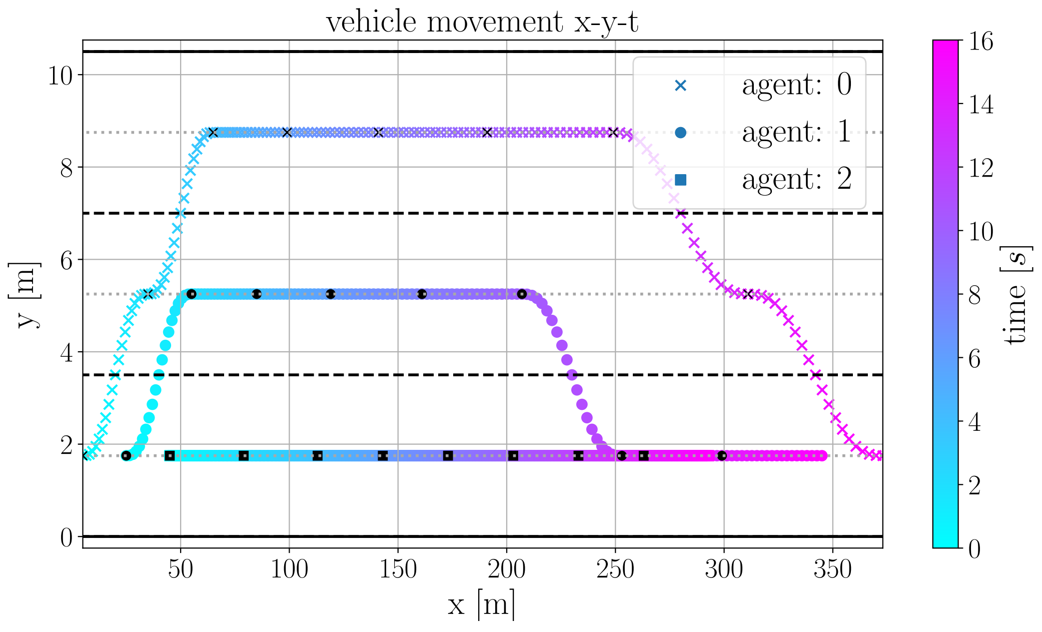


Figure 6.9.: 2D trajectories of each agent in scenario overtaking between 3 Vehicles. Different point types represent agents. Color of the points determines time according to the color bar. Agent 0 makes two left change to overtake the front two vehicles while agent 1 only changes once.

Table 6.5.: Independent variables and their values in the analysis of learning speed

variables	**values**
scenarios	*free drive*, *merge*, *double merge*, *overtaking* (2/3 vehicles), *bottleneck*
number of iterations	10, 20, 40, 60, 80, 100, 200, 400, 800, 1000, 2000, 6000
maximal search depth	5, 10, 15, 20, 30

search depth in one iteration including selection and simulation but the finally built tree doesn't necessarily reaches to this depth. Their values are shown in Table 6.5. Initial settings in each scenario can be found in A.2. For each configuration, we run the algorithm 15 times with varied random seeds.

6.3.2.3. Result Analysis

The results of all scenarios are listed in Fig. 6.10 to Fig. 6.14 with left column representing classical MCTS and right DecH-MCTS. $r_{collision}$ and $r_{success}$ are set as -100 and 100 respectively. Each data point has an error bar consisting of the upper and lower quartiles.

Generally speaking, classical MCTS and DecH-MCTS converges to an equilibrium after about 300 iterations in all scenarios. Both variants can generate solutions of same quality in simple scenarios, such as *free drive* and *merge*, and the former performs better when the iterations are very small (10 - 20). However, classical MCTS fails in complex scenaios, as we can see that DecH-MCTS converges to a better equilibrium as the number of iterations increases (> 100).

The reason is that too small amount of iterations results in ineffective learning of macro-actions in DecH-MCTS, while classical MCTS can find a relatively *safer* but not optimal

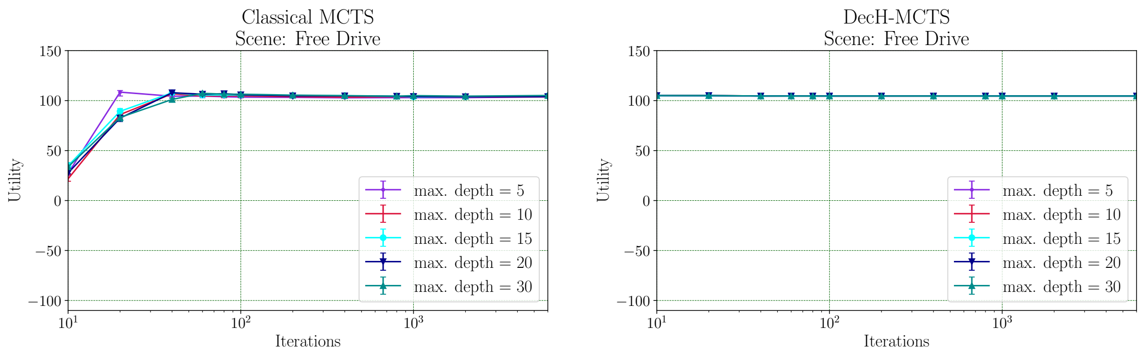


Figure 6.10.: Comparison of utility in scenario Free Drive

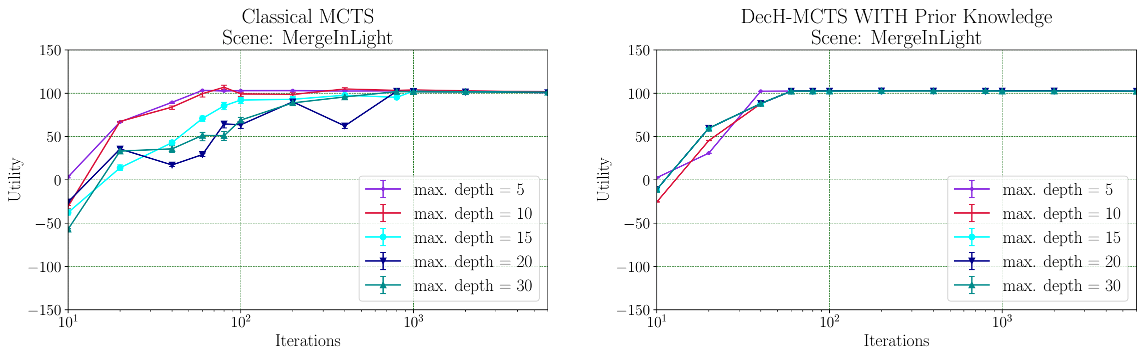


Figure 6.11.: Comparison of utility in scenario Merge

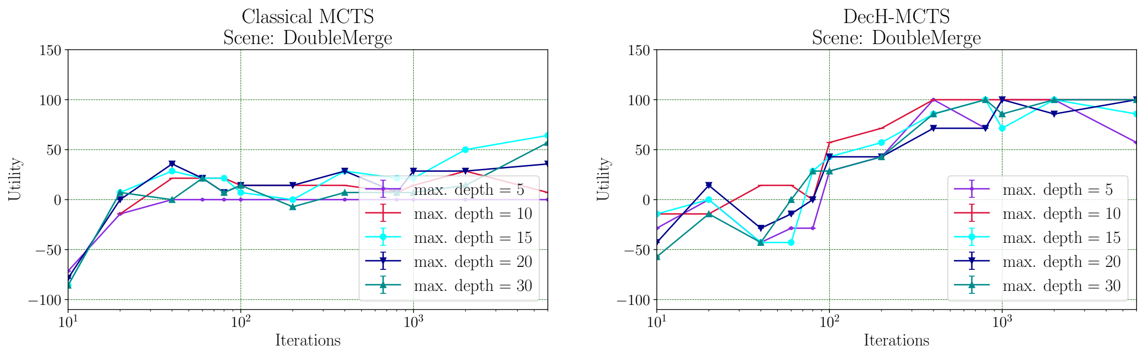
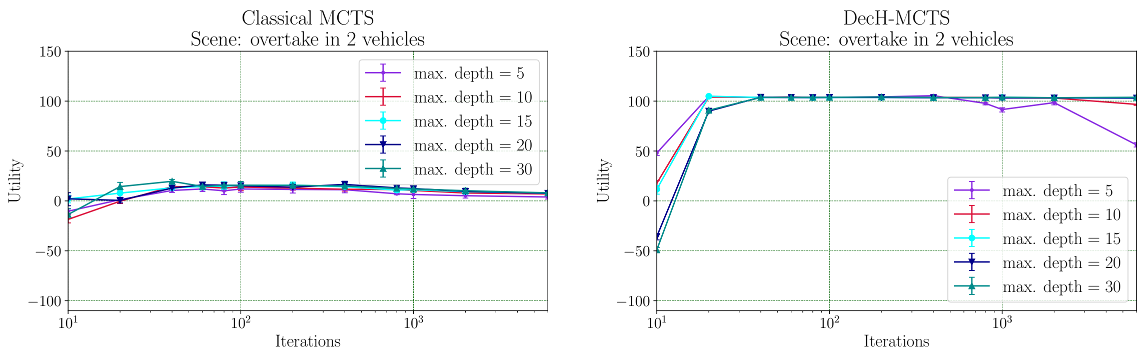
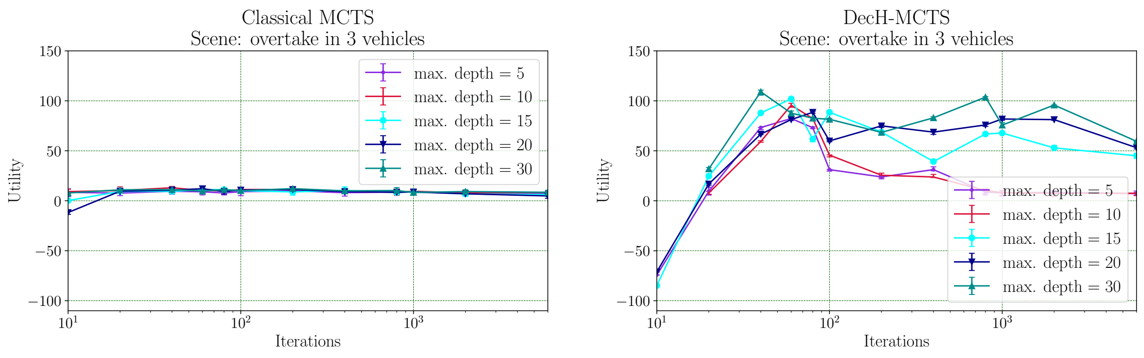


Figure 6.12.: Comparison of utility in scenario Double Merge



(a)Comparison of utility in scenario Overtaking between 2 Vehicles



(b)Comparison of utility in scenario Overtaking between 3 Vehicles

Figure 6.13.: Comparison of utility in scenario Overtaking between 2/3 Vehicles

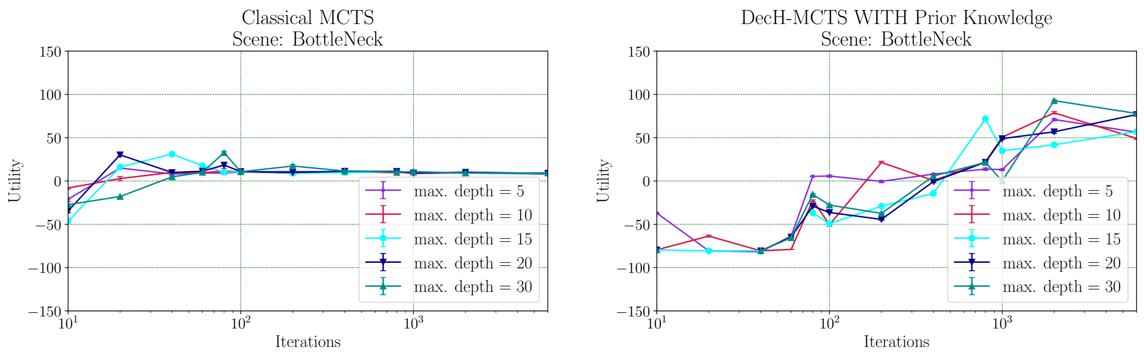


Figure 6.14.: Comparison of utility in scenario Bottleneck

action as long as the agent has explored all available actions (only five primitive actions). As more iterations are available, agents benefit from learning macro-actions and build a deeper search tree. By checking the generated solutions by classical MCTS, we find that classical MCTS tends to generate "safer" maneuvers, such as *deceleration*, *do nothing*. In the *double merge* scenario, vehicles even stop to keep large enough distance to others. Considering that the number of iterations usually lies around 1000, DecH-MCTS outperforms classical MCTS in practice.

In both *overtaking* scenarios, The most front vehicle controlled by classical MCTS firstly accelerates to make enough room for the behind vehicles' acceleration. The faster agents keep staying behind the slower one. The reason is that the random selection among all available actions in the simulation phase of classical MCTS results in bad evaluation of those actions which reduce the distance between vehicles and are more likely to cause collisions, while the available actions in DecH-MCTS are restricted to the child actions of parent actions. Additionally, the utility curves of DecH-MCTS in both *overtaking* scenarios shows that restricting the maximal search depth yields negative influence on the optimality because the macro-actions are too often interrupted when reaching the maximal search depth and the learning process is negatively influenced.

Moreover, in scenario *double merge* we can see that increasing the maximal search depth leads poorer performance of classical MCTS but the optimality of DecH-MCTS is not significantly influenced at larger maximal search depth. The reason is that classical MCTS does not have any domain specific knowledge and conducts the simulation phase totally randomly. Larger max. search depth means that the random simulation needs to run for more steps and very likely ends with collision. As a result, almost all actions are evaluated to be very bad since the simulation always returns a collision. It becomes difficult for the agent to distinguish the value of each action, thus resulting in solutions with lower quality.

6.3.3. Robustness

Considering the assumptions on homogeneous agents that we made when building the search tree, it is necessary to test if our algorithm can also produce feasible plans in the heterogeneous environment. This section demonstrates the performance of the DecH-MCTS algorithm for the scenarios *bottleneck* and *open loop* with heterogeneous agents.

6.3.3.1. Test Environment

To introduce heterogeneous agents, SUMO, an open source traffic simulation software developed by the DLR [KEBB12], is adopted. A screenshot of the SUMO gui is given in Fig. 6.15.

SUMO provides a C++ based interface named `TraCI` so that the target vehicle(s) can be controlled by extern applications. To realize the communication between our algorithm and SUMO, we build two ROS nodes, one for our planning algorithm, the other serves as interface which uses functions of `TraCI`, and uses the publish and subscribe functions to exchange the necessary messages. The information flow is visualized in Fig. 6.16.

Before the planning algorithm starts, the current state of the vehicles and the road information are sent by the node `SUMO interface`. The subscriber in the node `DecH-MCTS` receives the information and updates the current state. The planning is then executed based on the updated state and the best action is chosen. Corresponding trajectory is generated and sent to `SUMO interface`. With the help of the functions `MoveToXY` of `TraCI`, the target vehicle in SUMO moves according to the given trajectory. After execution, the environment is updated again and a new planning cycle starts.

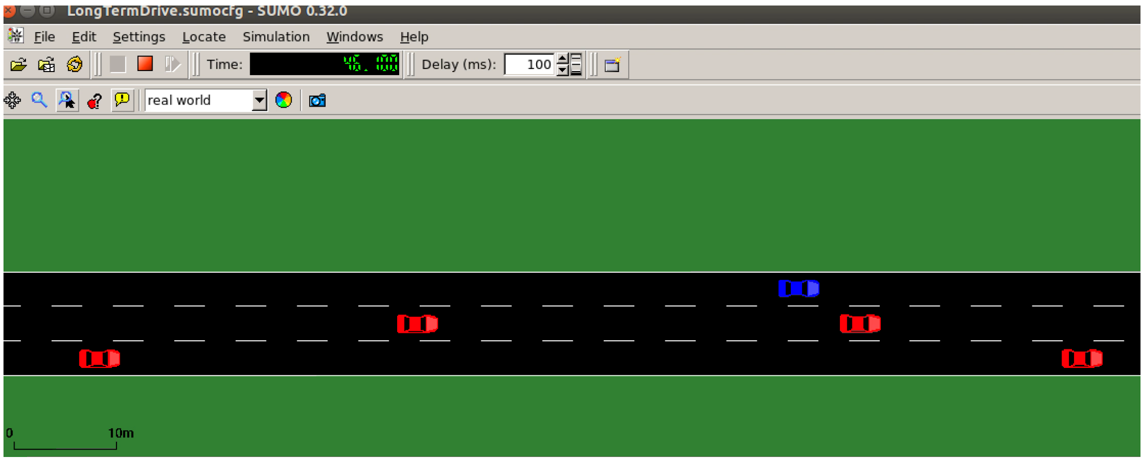


Figure 6.15.: Screenshot of SUMO-GUI

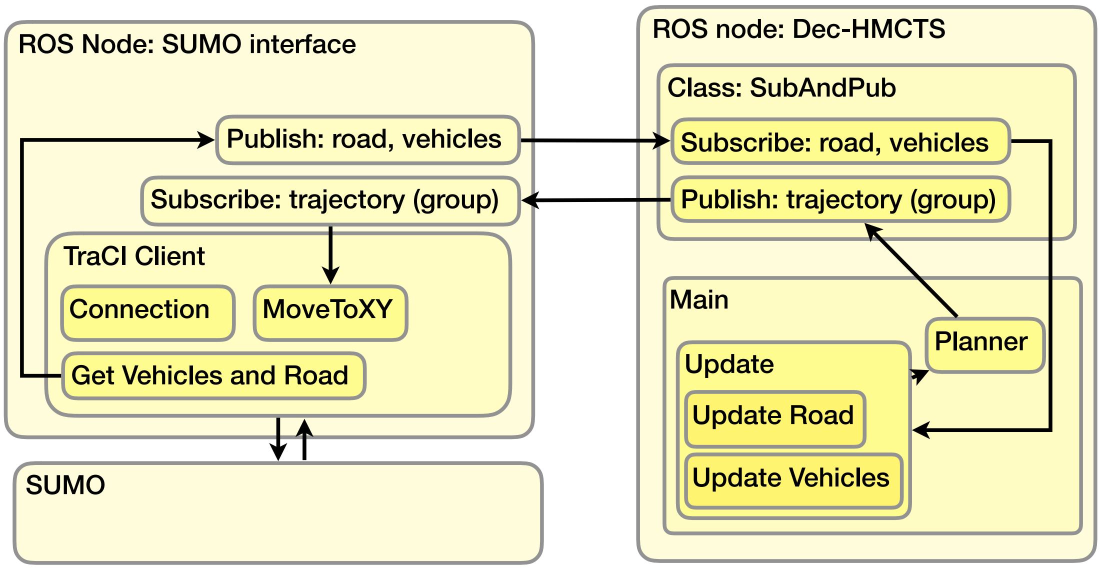


Figure 6.16.: Information Exchange between the planning algorithm and SUMO with the help of ROS-nodes

6.3.3.2. Adjustment in DecH-MCTS

In the practice we find that our algorithm tends to generate behaviors which are heavily dependent on others' cooperation, even if the cooperation factor λ of other agents is set to 0. As a result, collision often happens because other agents do not provide the expected cooperation. Through multiple experiments we find that the collision rate can be significantly reduced if we limit the action space of other agents when we build the tree. More specifically, the assumed action space of other agents is constrained to be the *make room* alone, which means that the ego agent assumes that other agents do not make any lane changes. By observing the behaviors of the vehicles controlled by SUMO (SUMO-vehicles), we find the reason for this improvement: the SUMO-vehicles seldom change lane to another one and prefer acceleration or deceleration to avoid the collisions with others. Consequently, if we assume that other agents' action space is limited to *make room* during the planning, the difference between the assumption and the reality is reduced so that the collision rate can be reduced.

In the following experiment, we preserve such adjustments on the modeling of other agents, i.e., assuming that others only keep its lane. Besides, the desired velocity of other agents v_d is set as current velocity and λ of them are all set as 0.

6.3.3.3. Uncooperative Bottleneck

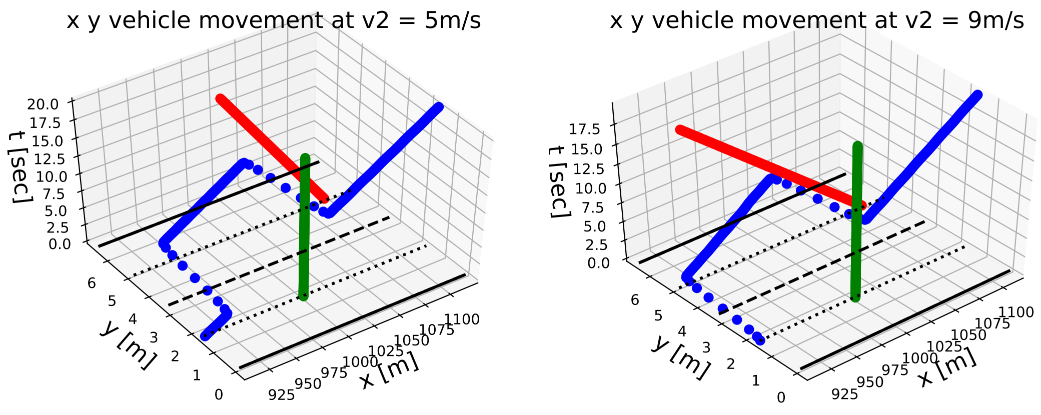
The scenario *Uncooperative Bottleneck* is designed to test if the algorithm is robust enough when other agents do not behave as the algorithm assumes (see Sec. 4.1.2.2). As Fig. 6.6 shows, agent 0 in blue comes from the left at $10m/s$ and is controlled by our algorithm, agent 1 in red comes from the right at different velocity ranging from 5m/s to 19m/s. Its behavior is predefined to keep constant velocity (*do nothing*). But the ego agent doesn't have this information. Besides, agent 2 in green stays static and blocks the lane of agent 0.

Fig. 6.17 shows the trajectories of three vehicles under each settings. When agent 1 drives at a slow speed, agent 0 chooses to drive faster to pass the bottleneck firstly. When agent 1 drives very fast, the agent 0 changes its plan according to the current situation and finally passes the bottleneck after agent 1. This shows that our algorithm is able to generate robust solutions in heterogeneous environment.

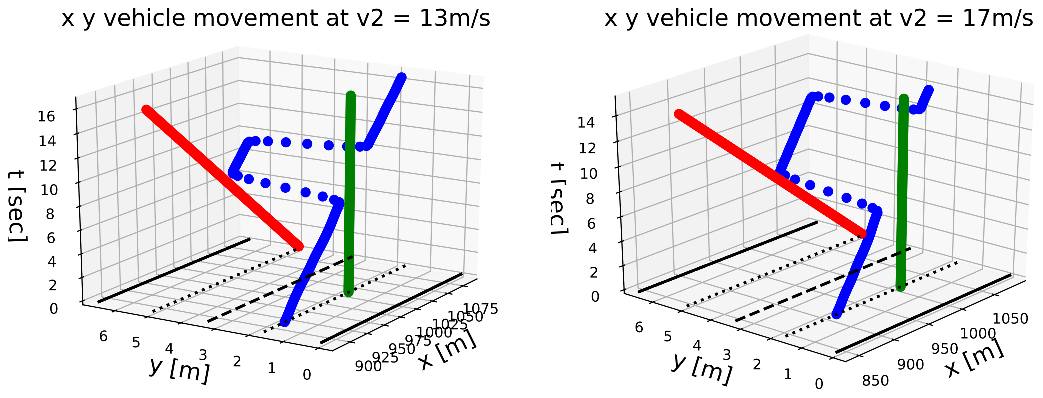
6.3.3.4. Open Loop Test

Driving in the open loop is another test of the robustness. The parameter setting is according to A.1. The ego agent (in blue) starts on lane 0 at $22m/s$ and has v_d of $40m/s$, l_d of 1. The desired velocity and desired lane of other agents v_d are both set as current value. Their λ are all set as 0. Four screenshots are shown in Fig. 6.18. The ego vehicle demonstrates rational behaviors, such as approaching the front vehicle to overtake, acceleration over the desired velocity to finish the overtaking maneuver, merging to the desired lane while keeping a safe distance with other vehicles, etc. Qualitative analysis shows that DecH-MCTS is able to generate feasible plans under heterogeneous environment where other agents do not always behave like the ego agent assumes.

To sum up, this section systematically evaluates the performance of our proposed algorithm DecH-MCTS in the cooperative automated driving problem. Experiments show that DecH-MCTS can solve multiple conflict scenarios without changing the parameter settings. Based on the hierarchically bounded return, the agent can flexibly learn the execution of macro-actions and how to choose one. The comparison with the classical MCTS shows that DecH-MCTS converges much faster than the classical MCTS and can find better equilibrium than classical MCTS. With modifications in the modeling of other agents our algorithm demonstrates good robustness under the heterogeneous environment.

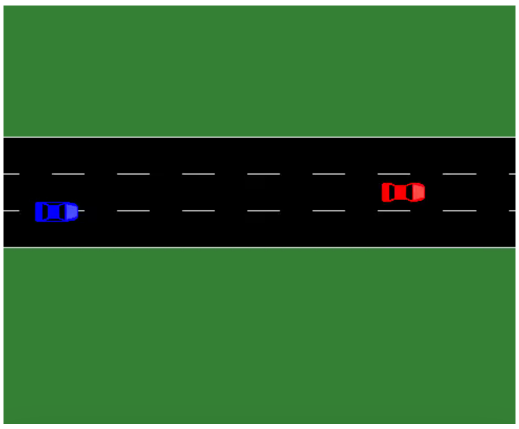


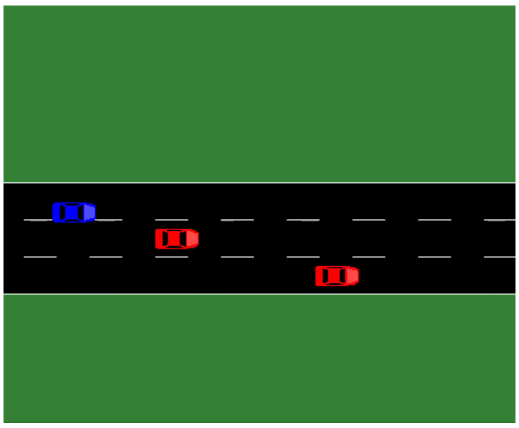
(a)Blue vehicle passes before the slow red one

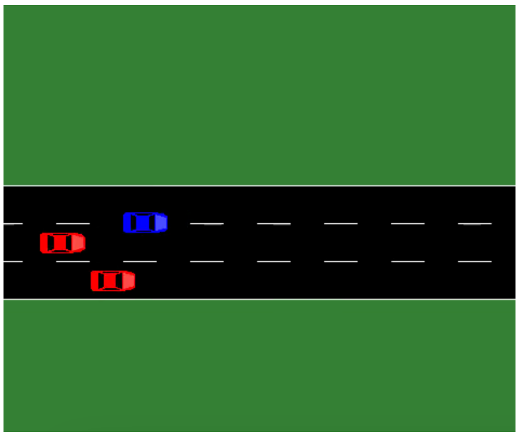


(b)Blue vehicle slows down and passes after the fast red one

Figure 6.17.: Analysis of robustness of DecH-MCTS in the scenario uncooperative bottleneck


(a) Ego agent merges to the desired lane.


(b) Ego agent changes to left to overtake the front vehicle.


(c) Ego agent finishes overtaking and merges to the desired lane.

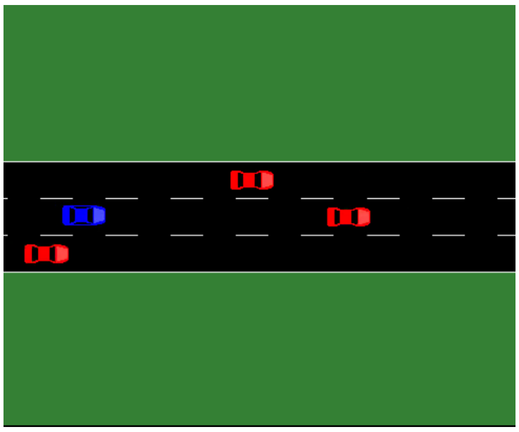

(d) Ego agent is approaching the front vehicle

Figure 6.18.: Screenshots from the Open Loop Test

7. Conclusion and Future Work

7.1. Conclusion

In this thesis, we propose a decentralized planning method with macro-actions based on MCTS to generate cooperative maneuvers with longer time horizons. Inspired by the hierarchical reinforcement learning framework *MAXQ*, we firstly propose four macro-actions: *overtake*, *merge in*, *make room* and *to desired velocity* based on real scenarios and design them as a hierarhical action graph. In order to realize the flexible design of macro-actions instead of filling them with if-else control sequences, we intensively study hierarchical reinforcement learning methods and propose the notion of *hierarhically bounded return*, which is described in detail using the example of MCTS with single agent. To extend to the decentralized multi-agent system, we adopt the $t_{continue}$ termination scheme for macro-actions to ensure asynchronous decision making and develop a hierarchical boundary check to realize the association and the calculation of the bounded return. Besides, the idea of *observational reinforcement* is adopted to incorporate the estimated others' reward into the local reward so that the agent can behave cooperatively.

The DecH-MCTS algorithm realizes simultaneously learning of the policies within and over macro-actions in a decentralized multi-agent system. The test scenario *overtaking* between 3 vehicles shows that two types of *overtake* are learned together with the policy of how to choose one macro-action. Our algorithm also demonstrates great generalization ability considering it can solve multiple conflict scenarios without changing the parameters. The comparison between classical MCTS and DecH-MCTS shows that DecH-MCTS outperforms classical MCTS in terms of convergence speed and planning quality. The robustness is evaluated in the test scenario uncooperative bottleneck and the open loop test with the help of SUMO. By introducing small modifications on the assumed action space and cooperation factor of other agents, our algorithm demonstrates robustness in heterogeneous environments.

7.2. Future Work

The DecH-MCTS algorithm shows promising result in achieving a faster and longer plan than the classical MCTS; however, some challenges still need to be investigated in the future.

It is strongly suggested to abandon the assumption of perfect perception in the further development. Instead, Partially Observable MDP (POMDP) should be employed so that the estimation of other agents and planning of own actions can be systematically combined. This would also be helpful in the improvement of robustness since the uncertainty about others' state is explicitly modeled.

Besides, the learning and planning are still based on the exact position and speed values. Each agent needs to record the action values in a table. It is difficult to reuse the already learned macro-actions to accelerate the further search. To the author's best knowledge there exist two possibilities to address such problem: value function approximation and state abstraction. The former approximates the value function with a formula based on the feature vector, which is very useful when the state space is huge. The latter has the similar idea with the temporal abstraction and abstracts the state to a higher level. For example, [KBSZ14] proposed a planning method based on the semantic state space based on the spatial relation of two vehicles. It would be a very good practice to combine the state abstraction and the temporal abstraction as [BSR16] did.

Finally, the problem of the strongly discretized action space should be addressed. MCTS variants for continuous planning can be incorporated into this algorithm and further improve the flexibility of the learned macro-actions.

A. Appendix

A.1. Parameter Settings

Table A.1.: Values of the basic parameters

symbol	**usage**	**value**
t_{sim} [s]	step length of one planning cycle, also the duration of primitive actions	2.0
$\Delta\dot{x}_+$ [m/s]	longitudinal velocity change of primitive action acceleration +	+4.0
$\Delta\dot{x}_-$ [m/s]	longitudinal velocity change of primitive action deceleration -	-4.0
w_s	weight on the longitudinal acceleration	-0.5
w_d	weight on the lane change	-7.0
w_v	weight on the velocity deviation to the desired velocity	+4.0
w_l	weight on the lane deviation to the desired lane	+20.0
$r_{collision}$	penalty when the vehicle collides with other vehicle(s)	-1000
$r_{invalid}$	penalty when the vehicle drives out of road	-1000
γ	discount factor when calculating the potential based shaping term and the return	0.98
ϵ	balance between the prior knowledge and random exploration	0.3
C_p	weight on the exploration term in the UCT calculation	$\sqrt{2}$

The tuning process is based on the scenarios *free drive*, *overtaking* between 2 and 3 vehicles, *bottleneck* and *merge*. Some domain specific knowledge can be useful to avoid unnecessary experiments. Here the four fundamental aspects are listed.

- action towards the desired state should be rewarded positively.

  When the vehicle is slower than its desired velocity, the acceleration should intuitively be the best action. The local reward $r^{(i)}_{local}$ of *acceleration* must be positive so that the agent can have a higher evaluation on it instead of choosing *do nothing*. As a result the relation between w_s and w_v should be maintained so that $r^{(i)}_{local} > 0$ under such situation holds. The same idea also applies in the relation between w_d and w_{lane}, where changing to the desired lane should be rewarded positively.

- the action cost for one time lane change is modeled as more expensive than one time acceleration.
- the collision and invalidity penalty should lie within a certain range.

  The collision and invalidity penalty should be large enough to avoid dangerous behavior but also not too large, otherwise the backpropagated penalty would dominate the update Eq. 4.12 and cause that all actions are evaluated similarly bad.
- Too short planning cycle time t_{sim} is impossible for the lane change and also not enough for generating a feasible plan. Too long cycle time would cause the vehicle cannot react to changes in the environment soon enough.

A.2. Scenario Descriptions

The initial position and velocity as well as the desires of each scenario are listed in the following tables A.2, A.3, A.4, A.5, A.6 and A.7.

Table A.2.: Free Drive

ID	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	5	4	1	28	2

Table A.3.: Overtaking between 2 vehicles

ID	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	5	15	0	25	0
1	25	15	0	15	0

Table A.4.: Overtaking between 3 vehicles

ID	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	5	15	0	30	0
1	25	15	0	25	0
2	45	15	0	15	0

Table A.5.: Merge

ID	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	5	25	0	25	1
1	5	25	1	25	1
2	100	0	0	0	0

Table A.6.: Double Merge

ID	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	5	15	0	15	1
1	5	15	2	15	1
2	100	0	0	0	0
3	100	0	1	0	1
4	105	0	0	0	0
5	105	0	1	0	1

Table A.7.: Bottleneck

ID	x_{ini} [m]	v_{ini} [m/s]	l_{ini}	v_d [m/s]	l_d
0	905	10	0	15	0
1	1095	-15	1	-15	1
2	1000	0	0	0	0

A.3. Solutions found by DecH-MCTS

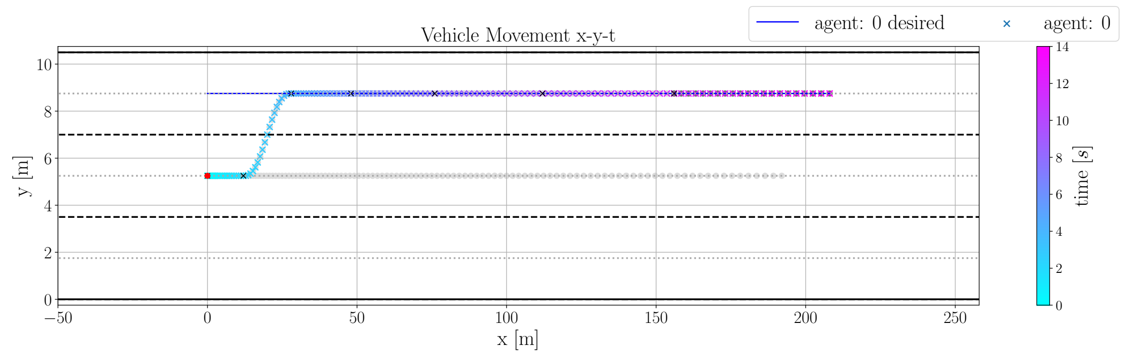
Here the solutions of all scenarios except the bottleneck (see 6.2) are listed in Fig. A.1, A.4, A.5, A.2 and A.3.

A.4. Computation Time

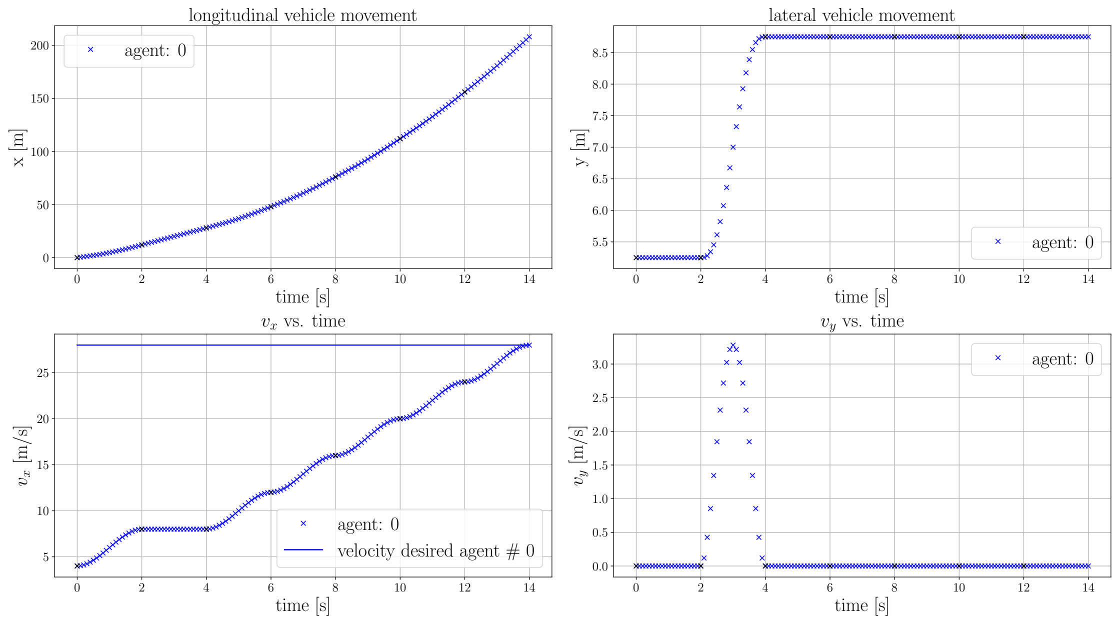
The evaluation is done on a virtual machine of Ubuntu 14.04 on a laptop Intel(R) Core(TM) i7 CPU at 2.6 GHz. The average computation time of one stage in different scenarios with different total iterations and maximal search depth (see Table 6.5) are listed in Figures A.6, A.7, A.8, A.9, A.10 and A.11. Each data point is the mean value of 15 repetitions and has an error bar consisting of the upper and lower quartiles.

A.5. Original Simulation Result in the Evaluation of Learning Speed

The collision rates and success rates for each scenario are listed and compared in Fig. A.12, A.15, A.16, A.14, A.13 and A.17.

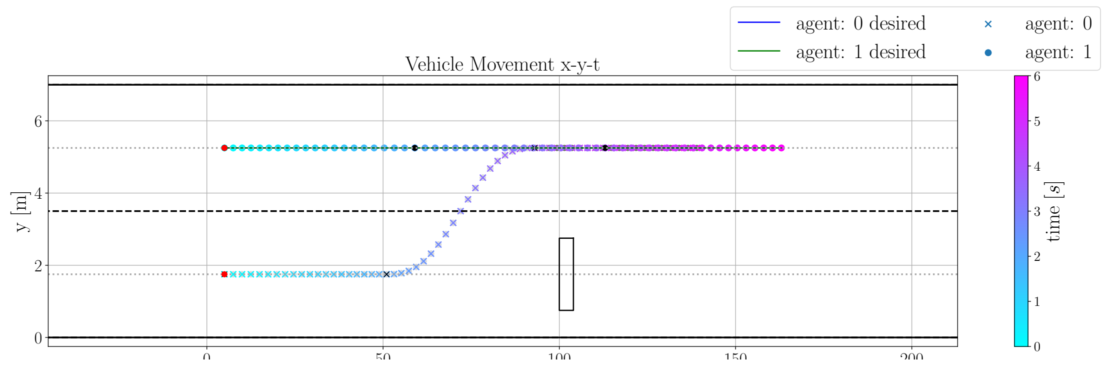


(a)2D Trajectory

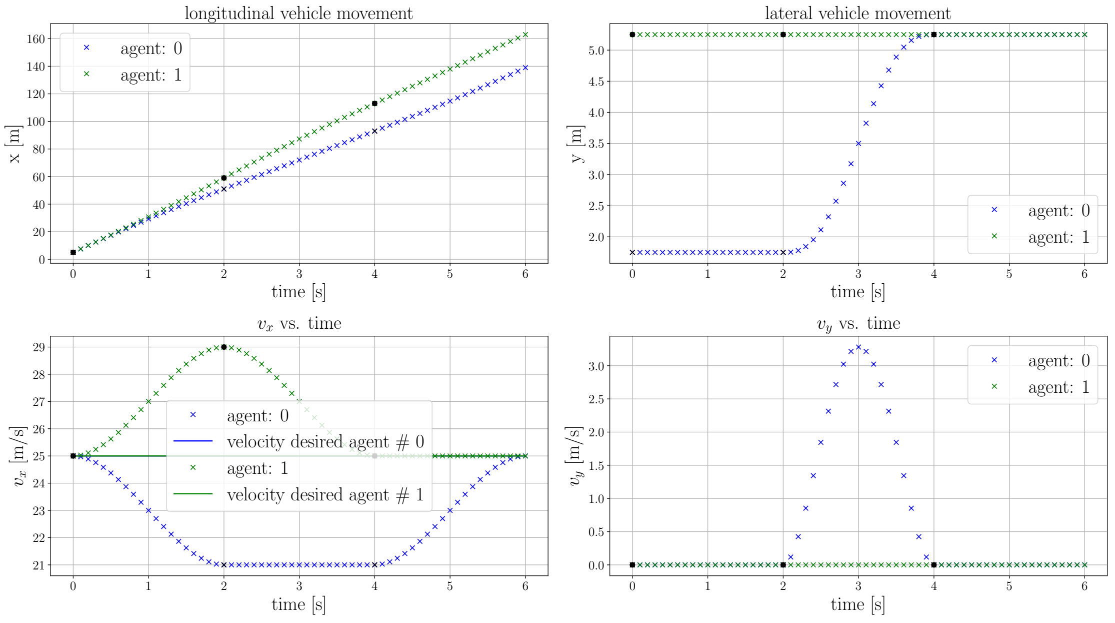


(b)Details

Figure A.1.: Solution for Scenario: Free Drive

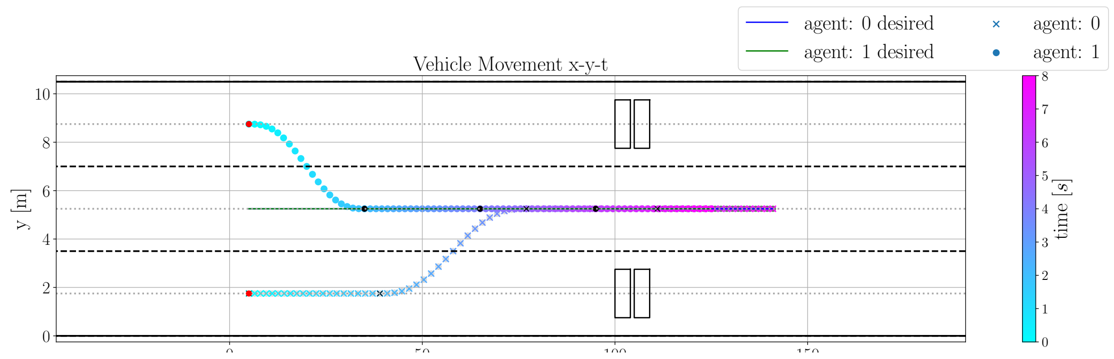


(a)2D Trajectory

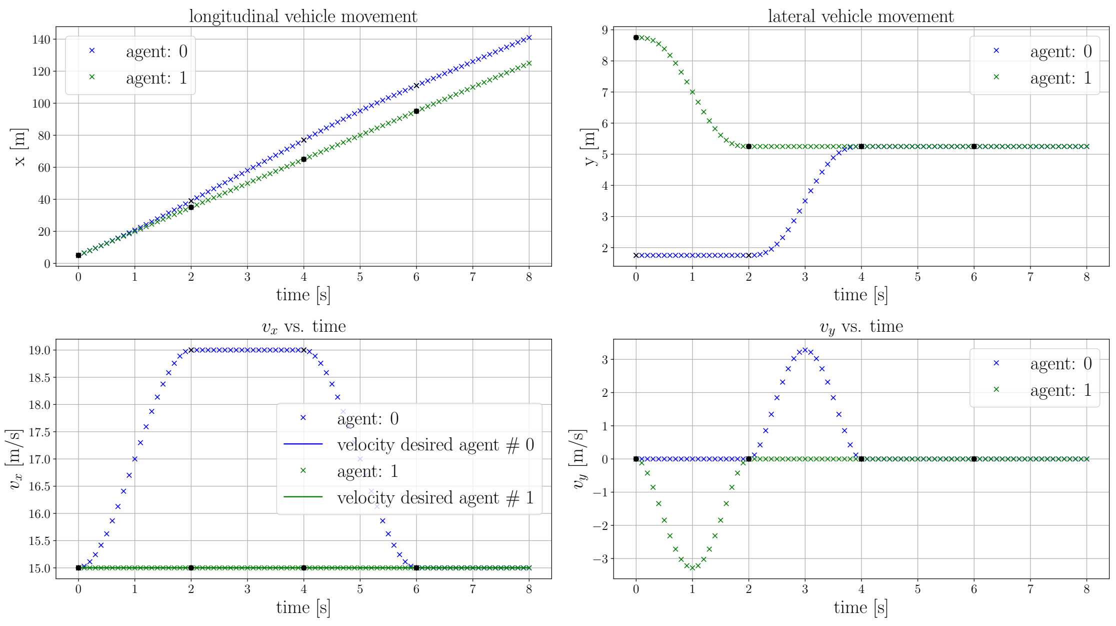


(b)Details

Figure A.2.: Solution for Scenario: Merge

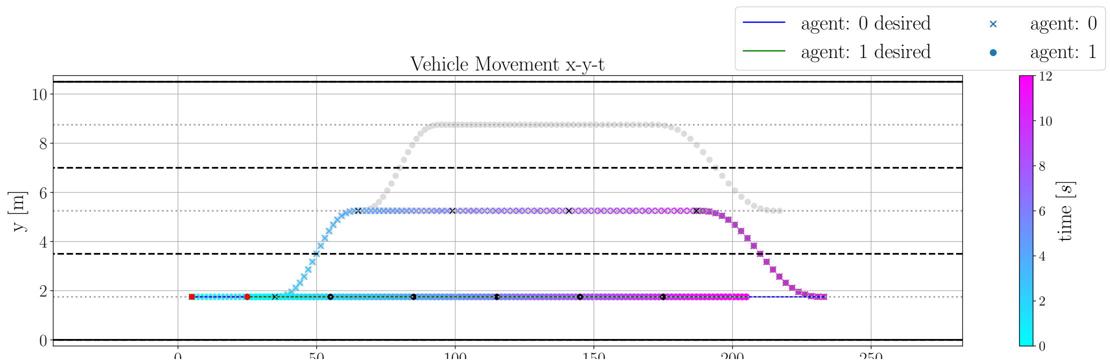


(a)2D Trajectory

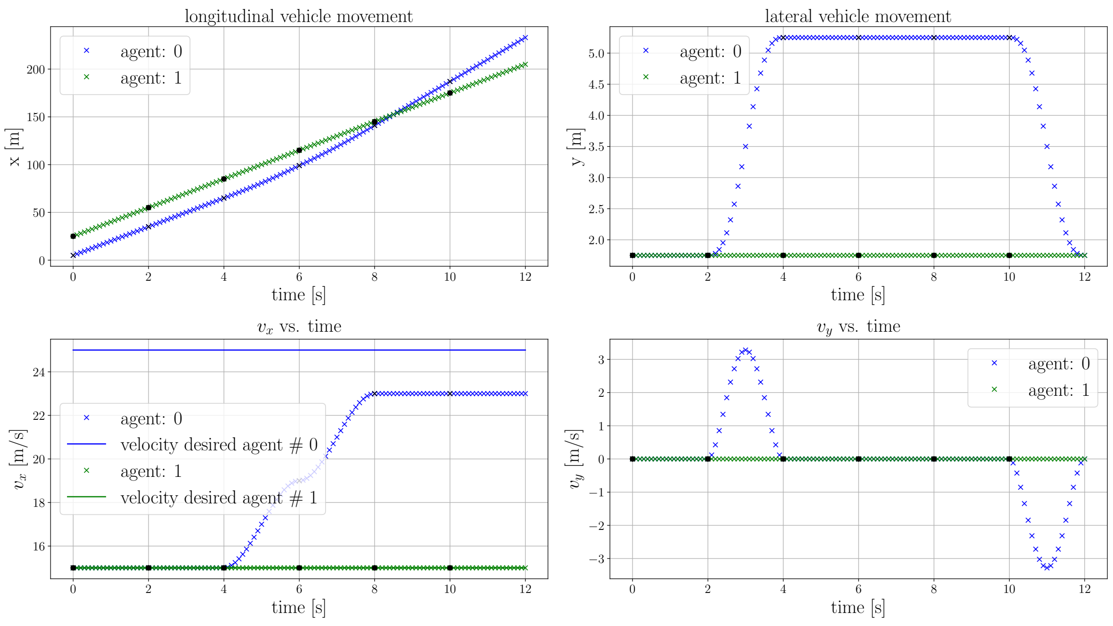


(b)Details

Figure A.3.: Solution for Scenario: Double Merge

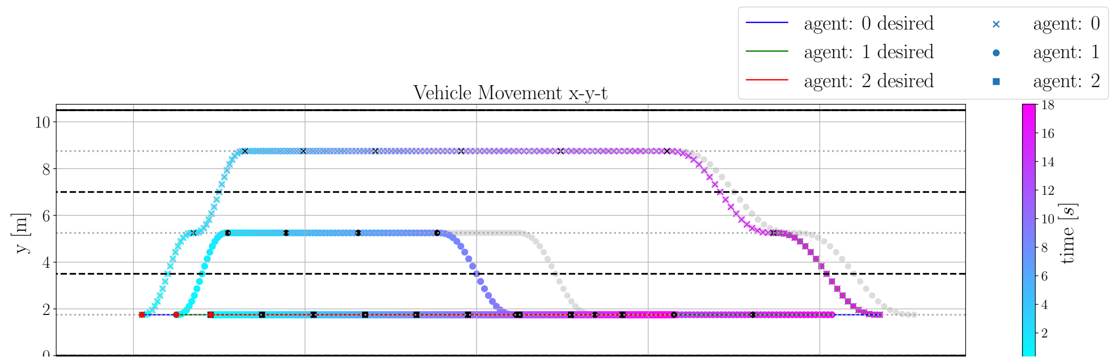


(a)2D Trajectory

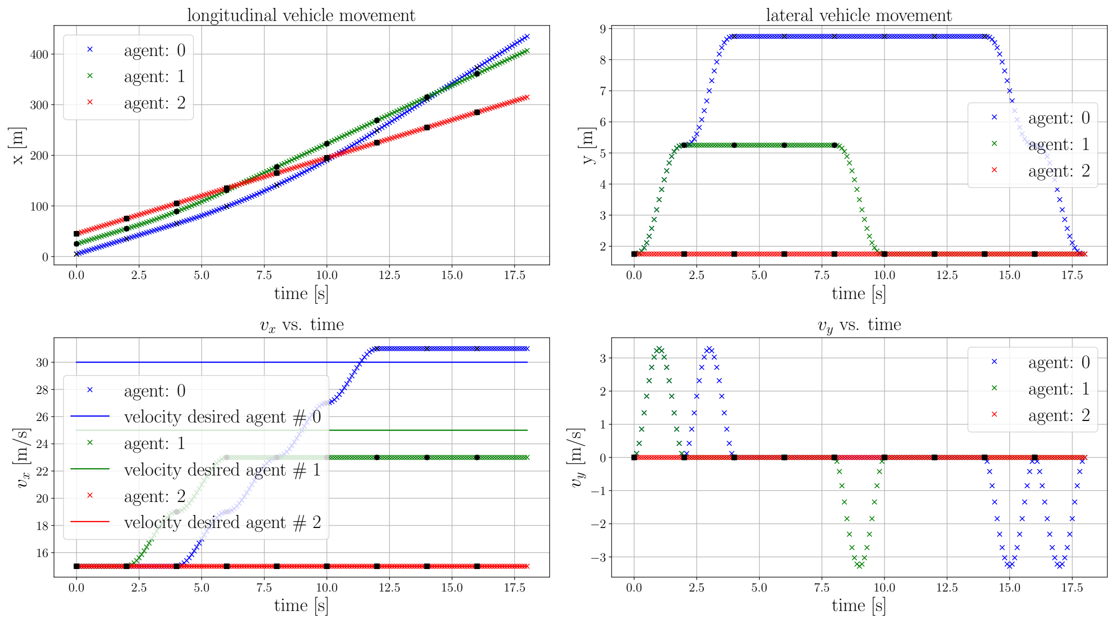


(b)Details

Figure A.4.: Solution for Scenario: Overtaking between 2 Vehicles



(a)2D Trajectory



(b)Details

Figure A.5.: Solution for Scenario: Overtaking between 3 Vehicles

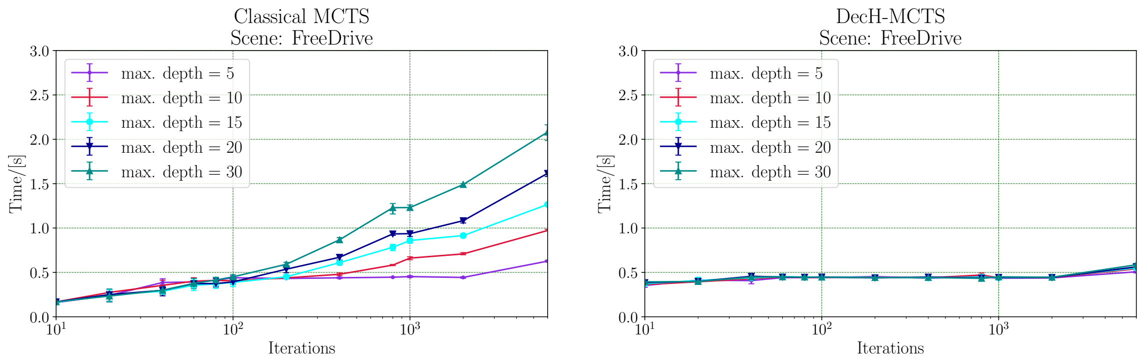


Figure A.6.: Comparison of Average Computation Time per Step in Scenario Free Drive

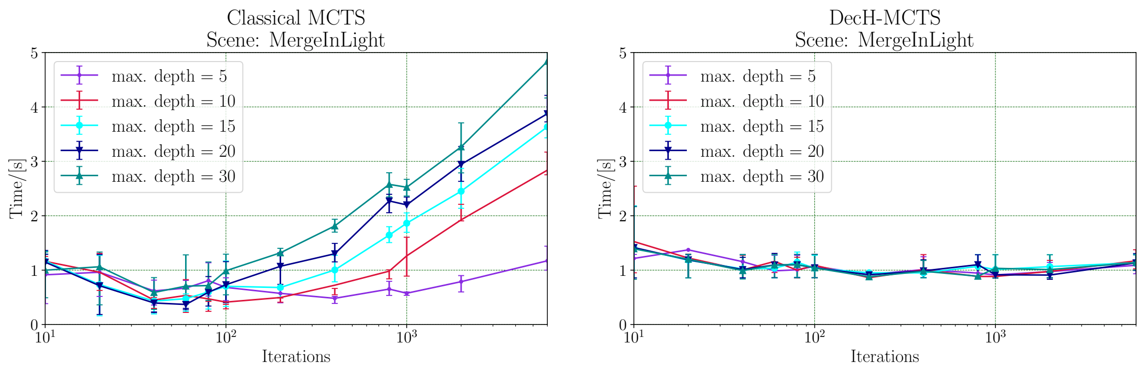


Figure A.7.: Comparison of Average Computation Time per Step in scenario Merge

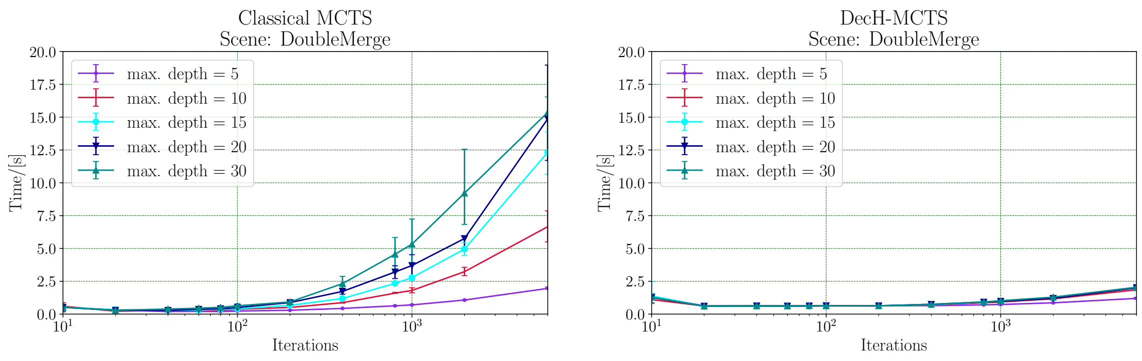


Figure A.8.: Comparison of Average Computation Time per Step in Scenario Double Merge

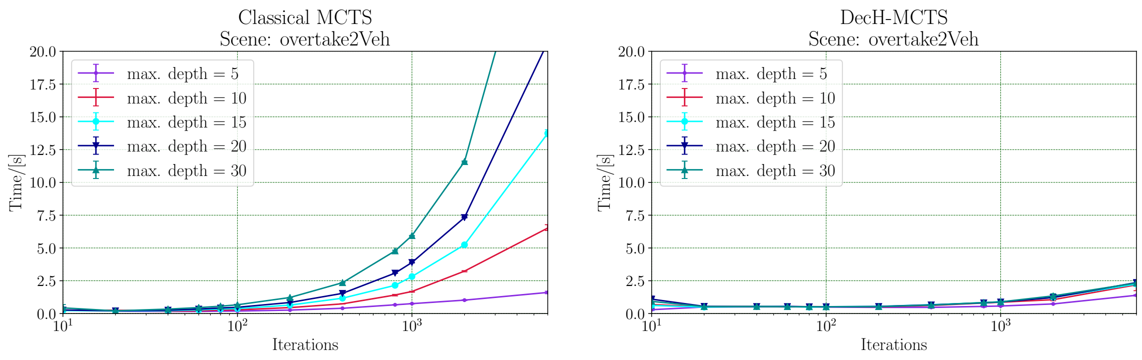


Figure A.9.: Comparison of Average Computation Time per Step in Scenario Overtaking between 2 Vehicles

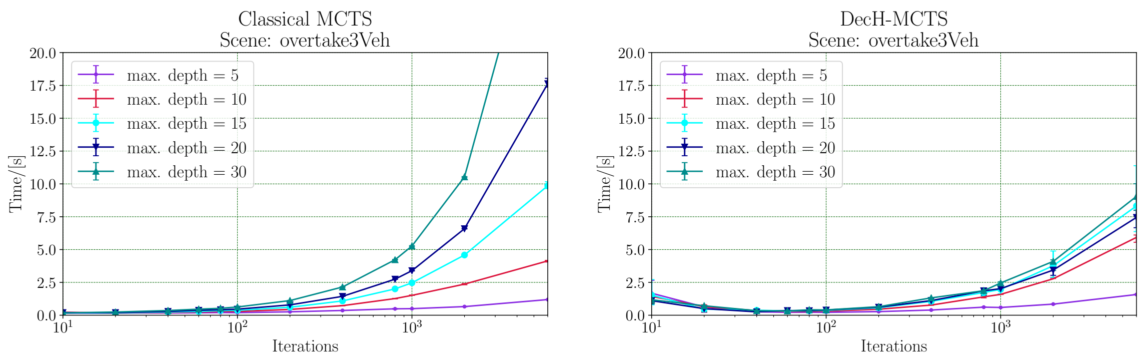


Figure A.10.: Comparison of Average Computation Time per Step in Scenario Overtaking between 3 Vehicles

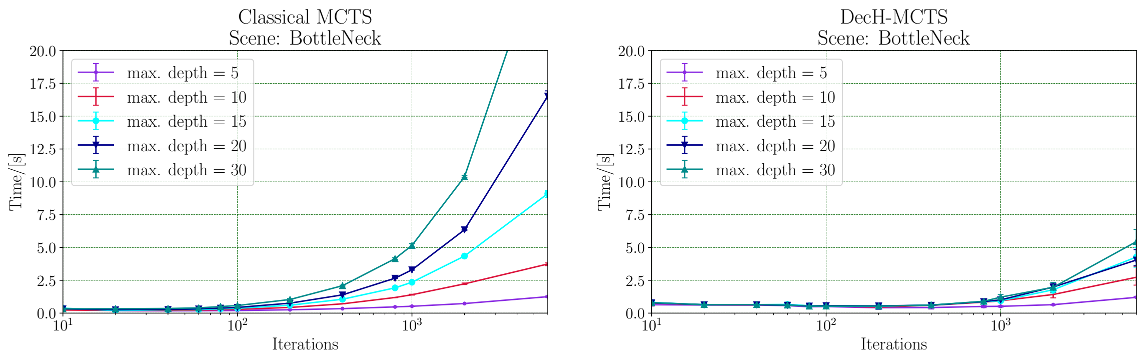
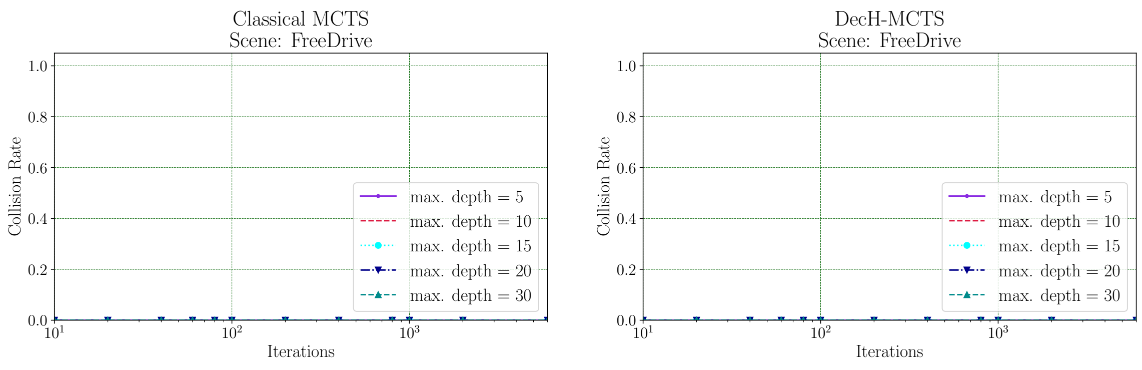
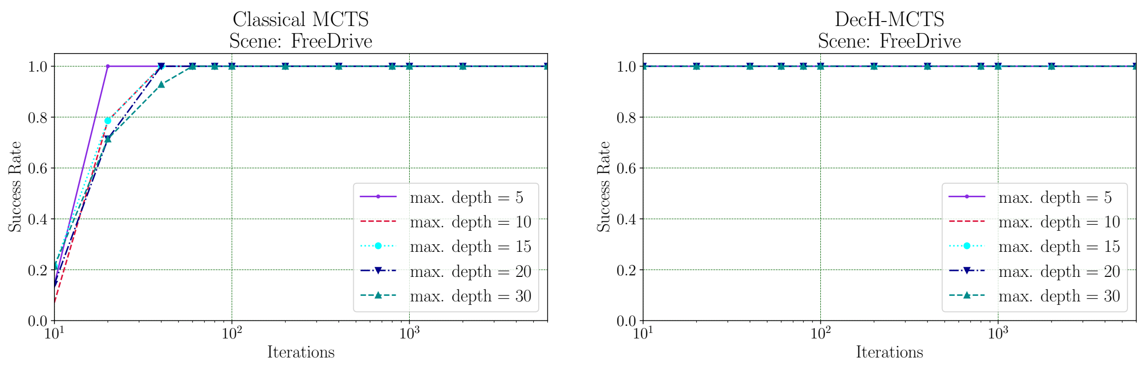


Figure A.11.: Comparison of Average Computation Time per Step in Scenario Bottleneck

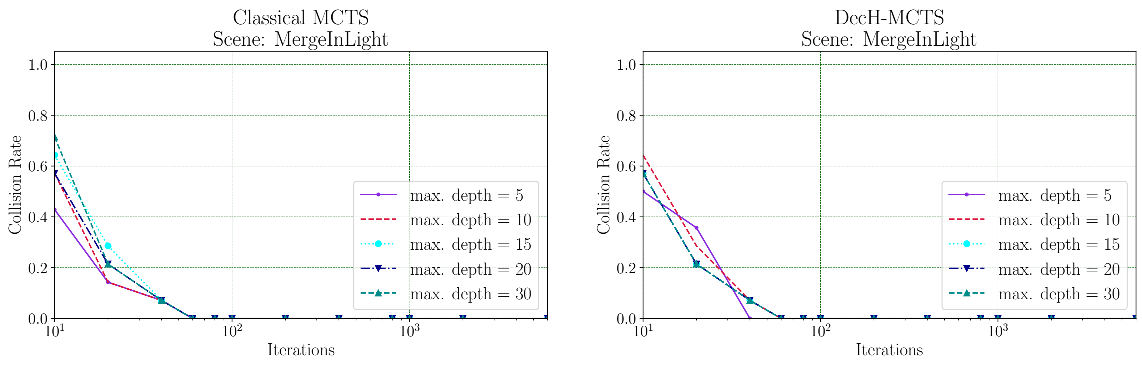


(a)Comparison of Collision Rates

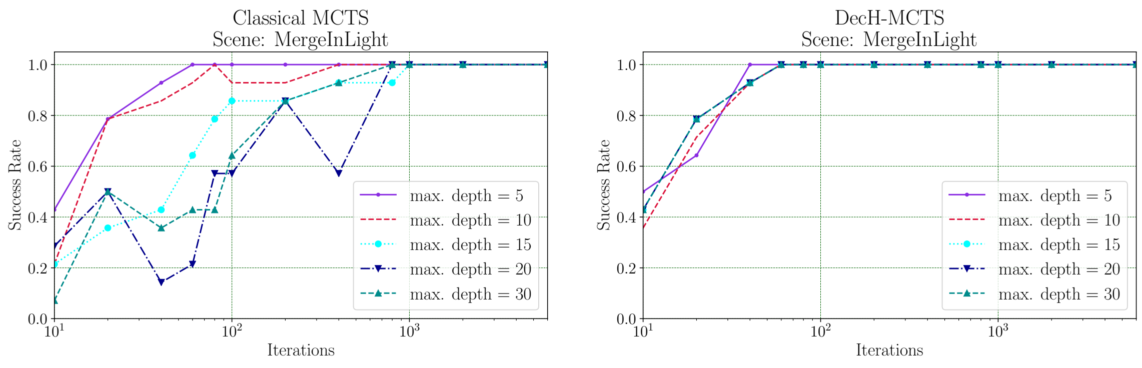


(b)Comparison of Success Rates

Figure A.12.: Performance comparison in Scenario Free Drive

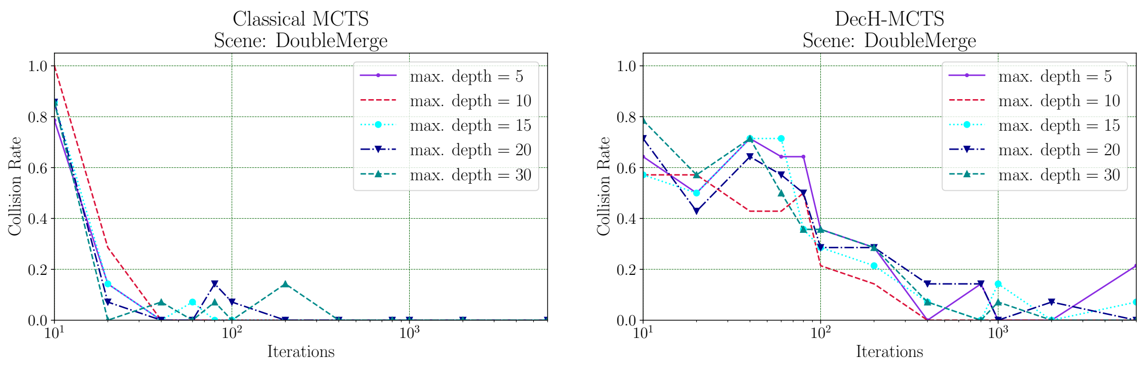


(a)Comparison of Collision Rates

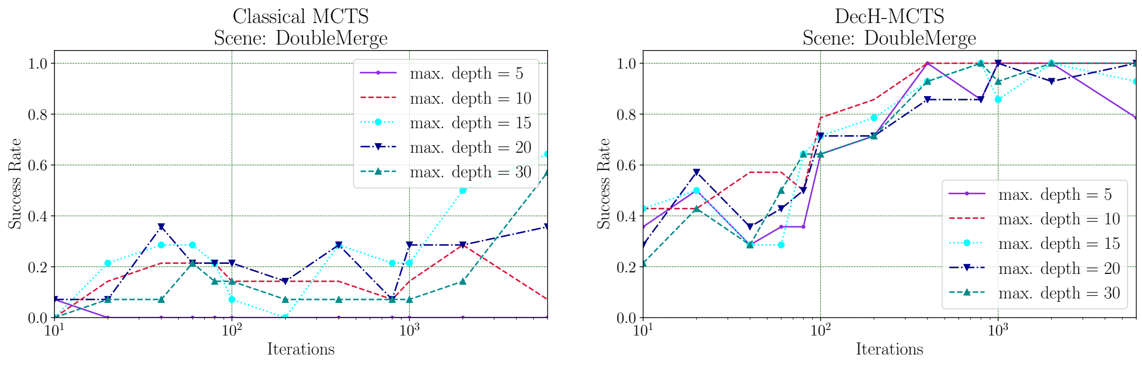


(b)Comparison of Success Rates

Figure A.13.: Performance comparison in Scenario Merge

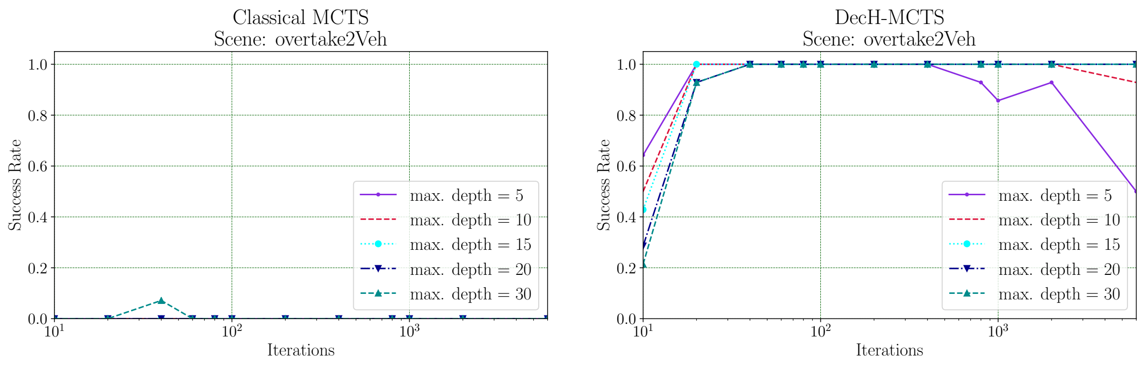


(a)Comparison of Collision Rates

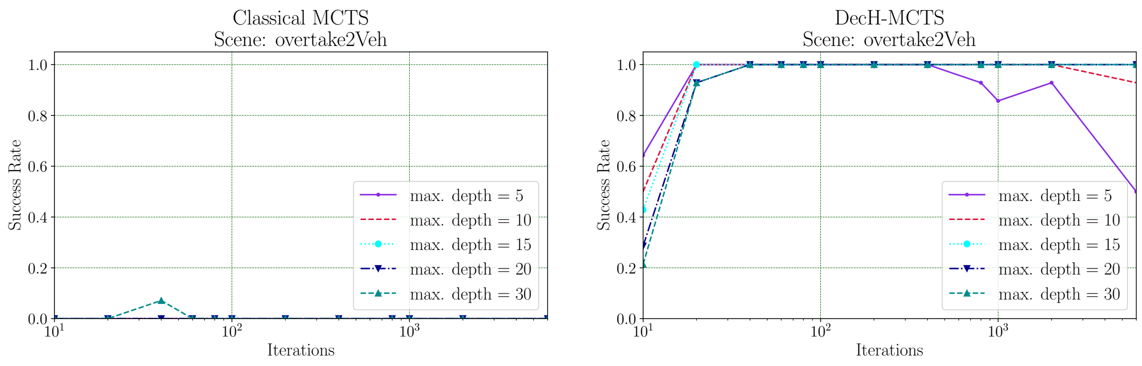


(b)Comparison of Success Rates

Figure A.14.: Performance comparison in Scenario Double Merge

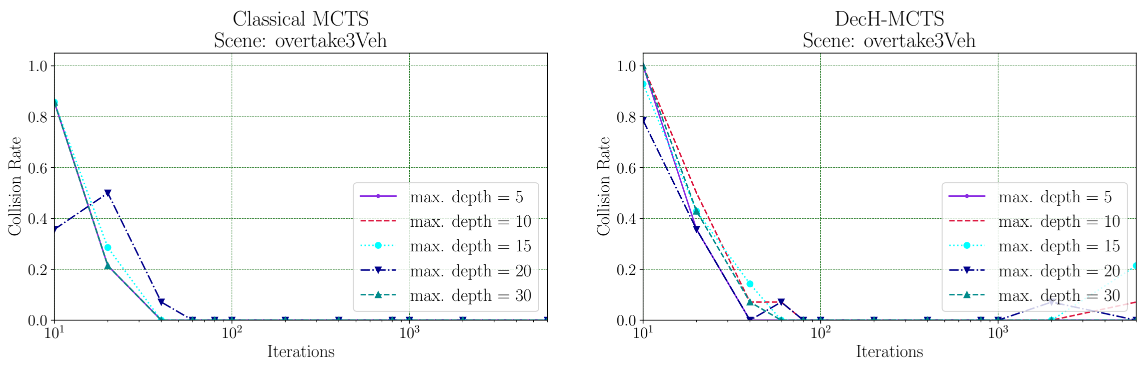


(a)Comparison of Success Rates

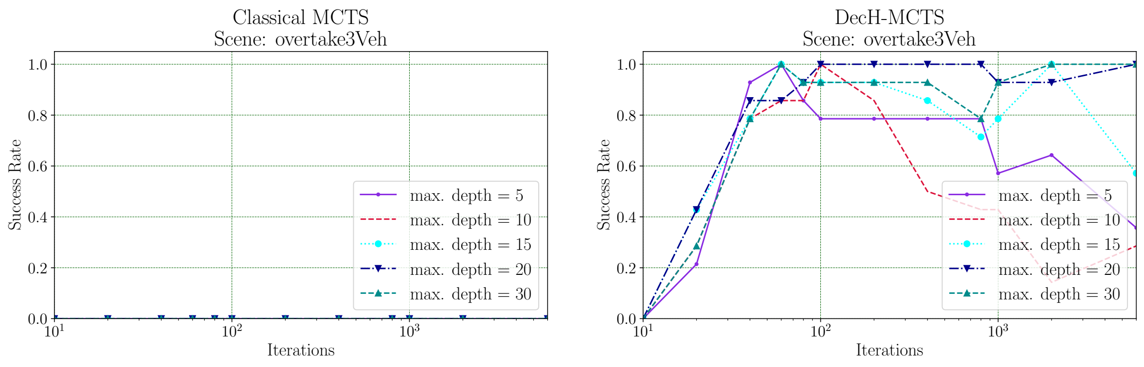


(b)Comparison of Success Rates

Figure A.15.: Performance Comparison in Scenario Overtaking between 2 Vehicles

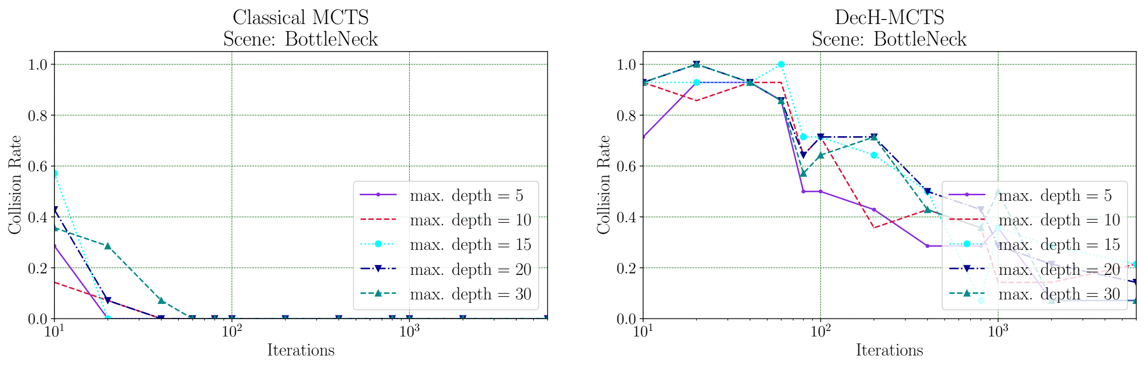


(a)Comparison of Collision Rates

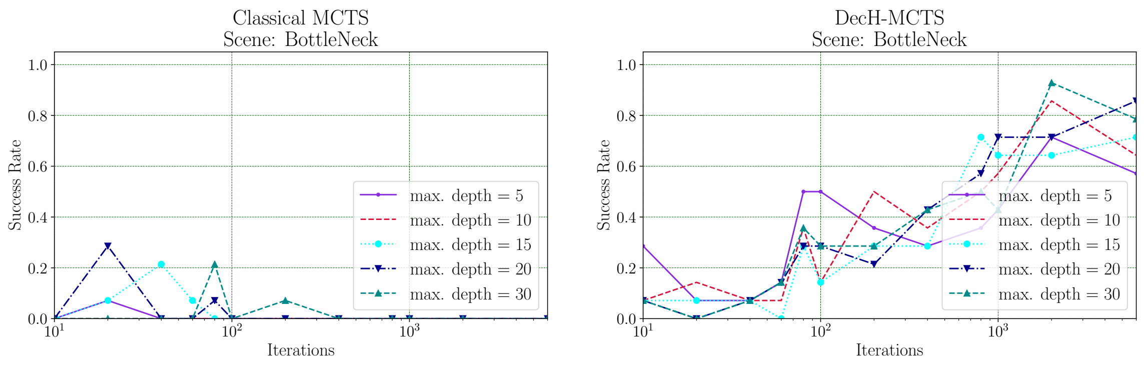


(b)Comparison of Success Rates

Figure A.16.: Performance Comparison in Scenario Overtaking between 3 Vehicles



(a)Comparison of Collision Rates



(b)Comparison of Success Rates

Figure A.17.: Performance Comparison in Scenario Bottleneck

List of Figures

2.1. Three termination schemes . . . 5
2.2. Learning and Planning . . . 6
2.3. Hierarchical Task Graph in the Taxi Domain . . . 8
2.4. Monte Carlo Tree Search in the example of an overtaking maneuver . . . 10
2.5. Matrix Game with 2 agents owning 2 primitive actions each . . . 11
2.6. Definition of Cooperative Behaviors . . . 12

4.1. Trajectories of the five primitive actions . . . 17
4.2. Update rule with example of a subtree starting from node 1 . . . 21
4.3. Hierarchical Action Graph . . . 22
4.4. Hierarchically Bounded Return in the example of single agent . . . 24

5.1. Hierarchically Bounded Return in an iteration of depth 7 . . . 32

6.1. Scenario: Free Drive with one vehicle . . . 36
6.2. Scenario: Merge; the green vehicle blocks one lane . . . 36
6.3. Scenario: Double Merge; the green vehicles block two lanes . . . 36
6.4. Scenario: Overtaking between 2 vehicles . . . 37
6.5. Scenario: Overtaking between 3 vehicles . . . 37
6.6. Scenario: Bottleneck; the green vehicle blocks one lane . . . 37
6.7. Scenario: Open Loop with heterogeneous environment . . . 38
6.8. Solution for scenario: Bottleneck . . . 39
6.9. 2D trajectories of each agent . . . 42
6.10. Comparison of utility in scenario Free Drive . . . 43
6.11. Comparison of utility in scenario Merge . . . 43
6.12. Comparison of utility in scenario Double Merge . . . 43
6.13. Comparison of utility in scenario Overtaking between 2/3 Vehicles . . . 44
6.14. Comparison of utility in scenario Bottleneck . . . 44
6.15. Screenshot of SUMO-GUI . . . 46
6.16. Information Exchange between the planning algorithm and SUMO with the help of ROS-nodes . . . 46
6.17. Analysis of robustness of DecH-MCTS in the scenario uncooperative bottleneck . . . 48
6.18. Screenshots from the Open Loop Test . . . 49

A.1. Solution for Scenario: Free Drive . . . 56
A.2. Solution for Scenario: Merge . . . 57
A.3. Solution for Scenario: Double Merge . . . 58
A.4. Solution for Scenario: Overtaking between 2 Vehicles . . . 59
A.5. Solution for Scenario: Overtaking between 3 Vehicles . . . 60
A.6. Comparison of Average Computation Time per Step in Scenario Free Drive 60
A.7. Comparison of Average Computation Time per Step in scenario Merge . . . 61

A.8. Comparison of Average Computation Time per Step in Scenario Double Merge 61
A.9. Comparison of Average Computation Time per Step in Scenario Overtaking between 2 Vehicles 61
A.10.Comparison of Average Computation Time per Step in Scenario Overtaking between 3 Vehicles 62
A.11.Comparison of Average Computation Time per Step in Scenario Bottleneck 62
A.12.Performance comparison in Scenario Free Drive 63
A.13.Performance comparison in Scenario Merge 63
A.14.Performance comparison in Scenario Double Merge 64
A.15.Performance Comparison in Scenario Overtaking between 2 Vehicles 64
A.16.Performance Comparison in Scenario Overtaking between 3 Vehicles 65
A.17.Performance Comparison in Scenario Bottleneck 65

List of Tables

3.1. Nomenclature 16

4.1. Initial and terminal conditions for macro-actions 22

6.1. Initial Condition for Scenario Overtaking between 3 vehicles 38
6.2. Planned action sequence at $t = 0$ 40
6.3. State after the first step (actions L, L, 0 are respectively executed) 40
6.4. Planned action sequence at $t = 2\,\mathrm{s}$ 41
6.5. Independent variables and their values in the analysis of learning speed . . 42

A.1. Values of the basic parameters 53
A.2. Free Drive 54
A.3. Overtaking between 2 vehicles 54
A.4. Overtaking between 3 vehicles 54
A.5. Merge 54
A.6. Double Merge 54
A.7. Bottleneck 55

Bibliography

[ACBF02] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[ACBFS95] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, “Gambling in a rigged casino: The adversarial multi-armed bandit problem,” in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on.* IEEE, 1995, pp. 322–331.

[AKK14] C. Amato, G. D. Konidaris, and L. P. Kaelbling, “Planning with Macro-Actions in Decentralized POMDPs,” *Proceedings of the 13th Inter- national Conference on Autonomous Agents and Multiagent Systems*, pp. 1273–1280, 2014.

[AKM17] M. Althoff, M. Koschi, and S. Manzinger, “Commonroad: Composable benchmarks for motion planning on roads,” in *Intelligent Vehicles Symposium (IV), 2017 IEEE.* IEEE, 2017, pp. 719–726.

[BD95] S. J. Bradtke and M. O. Duff, “Reinforcement learning methods for continuous-time markov decision problems,” in *Advances in neural information processing systems*, 1995, pp. 393–400.

[BF09] Y. Bjoernsson and H. Finnsson, “Cadiaplayer: A simulation-based general game player,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 1, pp. 4 – 15, 04 2009.

[BM03] A. G. Barto and S. Mahadevan, “Recent advances in hierarchical reinforcement learning,” *Discrete Event Dynamic Systems*, vol. 13, no. 4, pp. 341–379, 2003.

[BPW$^+$12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of Monte Carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, mar 2012.

[BSA83] A. G. Barto, R. S. Sutton, and C. W. Anderson, “Neuronlike adaptive elements that can solve difficult learning control problems,” *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[BSR16] A. Bai, S. Srivastava, and S. Russell, “Markovian state and action abstractions for MDPs via hierarchical MCTS,” in *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2016-Janua, 2016, pp. 3029–3037. [Online]. Available: https://www.ijcai.org/Proceedings/16/Papers/430.pdf

[CPW12] Cowling, Peter I., E. J. Powley, and D. Whitehouse, “Information set monte carlo tree search,” *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, vol. 4, no. 2, pp. 120–143, 2012.

[Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=6203567

[Die00] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *J. Artif. Intell. Res.(JAIR)*, vol. 13, pp. 227–303, 2000.

[DP14] M. During and P. Pascheka, "Cooperative decentralized decision making for conflict resolution among autonomous agents," in *Innovations in Intelligent Systems and Applications (INISTA) Proceedings, 2014 IEEE International Symposium on.* IEEE, 2014, pp. 154–161.

[DTW11] N. G. Den Teuling and M. H. Winands, "Monte-carlo tree search for the simultaneous move game tron," *Univ. Maastricht, Netherlands, Tech. Rep*, 2011.

[DU07] P. Drake and S. Uurtamo, "Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go," 01 2007.

[dWRB16] M. de Waard, D. M. Roijers, and S. C. Bakkes, "Monte carlo tree search with options for general video game playing," in *IEEE Conference on Computational Intelligence and Games (CIG).* IEEE, 2016, pp. 1–8.

[GMM$^+$06] M. Ghavamzadeh, S. Mahadevan, R. Makar, M. Ghavamzadeh, S. Mahadevan, and R. Makar, "Hierarchical multi-agent reinforcement learning," *Auton Agent Multi-Agent Sys*, vol. 13, pp. 197–229, 2006. [Online]. Available: https://link.springer.com/content/pdf/10.1007{%}2Fs10458-006-7035-4.pdf

[GS07] S. Gelly and D. Silver, "Combining online and offline knowledge in uct," in *Proceedings of the 24th international conference on Machine learning.* ACM, 2007, pp. 273–280.

[KBSZ14] R. Kohlhaas, T. Bittner, T. Schamm, and J. M. Zöllner, "Semantic state space for high-level maneuver planning in structured traffic scenes," in *Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference on.* IEEE, 2014, pp. 1060–1065.

[KEBB12] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent Development and Applications of {SUMO - Simulation of Urban MObility}," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3, pp. 128–138, 2012. [Online]. Available: http://www.iariajournals.org/systems{_}and{_}measurements/http://elib.dlr.de/80483/

[KLM96] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[KMN99] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large Markov decision processes," in *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2, 1999, pp. 1324–1331. [Online]. Available: https://www.cis.upenn.edu/{~}mkearns/papers/sparsesampling-journal.pdf

[KSW06] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved monte-carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[LKK16] D. Lenz, T. Kessler, and A. Knoll, "Tactical cooperative planning for autonomous highway driving using Monte-Carlo Tree Search," in *IEEE Intelligent Vehicles Symposium, Proceedings*, vol. 2016-Augus. IEEE, jun 2016, pp. 447–453. [Online]. Available: http://ieeexplore.ieee.org/document/7535424/

[LR00] M. Lauer and M. Riedmiller, “An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems,” in *17-th International Conference on Machine Learning*, 2000, pp. 535–542. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.772

[LSO$^+$17] M. Liu, K. Sivakumar, S. Omidshafiei, C. Amato, and J. P. How, “Learning for Multi-robot Cooperation in Partially Observable Stochastic Environments with Macro-actions,” 2017. [Online]. Available: http://arxiv.org/abs/1707.07399

[Mat94] M. J. Mataric, “Learning to behave socially,” in *Third international conference on simulation of adaptive behavior*, vol. 617, 1994, pp. 453–462.

[MKS$^+$13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[MLA17] S. Manzinger, M. Leibold, and M. Althoff, “Driving strategy selection for cooperative vehicles using maneuver templates,” in *Intelligent Vehicles Symposium (IV), 2017 IEEE.* IEEE, 2017, pp. 647–654.

[NHR99] A. Y. Ng, D. Harada, and S. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping,” in *ICML*, vol. 99, 1999, pp. 278–287.

[NOB$^+$17] M. Naumann, P. Orzechowski, C. Burger, O. Tas, and C. Stiller, “Herausforderungen für die verhaltensplanung kooperativer automatischer fahrzeuge,” *AAET Automatisiertes und vernetztes Fahren. Braunschweig, Germany: ITS automotive nord eV*, pp. 287–307, 2017.

[NS17] M. Naumann and C. Stiller, “Towards cooperative motion planning for automated vehicles in mixed traffic,” *arXiv preprint arXiv:1708.06962*, 2017.

[Omi15] Omidshafiei, Shayegan and Agha-Mohammadi, Ali-Akbar and Amato, Christopher and How, Jonathan P, “Decentralized control of partially observable markov decision processes using belief space macro-actions,” in *Robotics and Automation (ICRA), 2015 IEEE International Conference on.* IEEE, 2015, pp. 5962–5969.

[PPW$^+$14] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. M. Lucas, “Solving the physical traveling salesman problem: Tree search and macro actions,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 31–45, mar 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6517274/

[PR98] R. Parr and S. J. Russell, “Reinforcement learning with hierarchies of machines,” in *Advances in neural information processing systems*, 1998, pp. 1043–1049.

[Pre00] D. Precup, *Temporal abstraction in reinforcement learning.* University of Massachusetts Amherst, 2000.

[PRHK17] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov, “Combining Neural Networks and Tree Search for Task and Motion Planning in Challenging Environments,” mar 2017. [Online]. Available: http://arxiv.org/abs/1703.07887

[PWC12] E. J. Powley, D. Whitehouse, and P. I. Cowling, “Monte Carlo Tree Search with macro-actions and heuristic route planning for the Physical Travelling

Salesman Problem," in *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*. IEEE, sep 2012, pp. 234–241. [Online]. Available: http://ieeexplore.ieee.org/document/6374161/

[RM03] K. Rohanimanesh and S. Mahadevan, "Learning to take concurrent actions," in *Advances in neural information processing systems*, 2003, pp. 1651–1658.

[SB16] R. S. Sutton and A. G. Barto, *Reinforcement learning : an introduction.* MIT Press, 2016.

[SFK07] C. Stiller, G. Farber, and S. Kammel, "Cooperative cognitive automobiles," in *Intelligent Vehicles Symposium, 2007 IEEE.* IEEE, 2007, pp. 215–220.

[SH96] D. Swaroop and J. K. Hedrick, "String stability of interconnected systems," *IEEE transactions on automatic control*, vol. 41, no. 3, pp. 349–357, 1996.

[SHM$^+$16a] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, jan 2016. [Online]. Available: http://www.nature.com/doifinder/10.1038/nature16961

[SHM$^+$16b] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[SPS99] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[SSS09] M. Shaei, N. Sturtevant, and J. Schaeffer, "Comparing UCT versus CFR in simultaneous games," in *IJCAI Workshop on General Game Playing*, 2009, pp. 75–82.

[SSS$^+$17a] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, oct 2017. [Online]. Available: http://www.nature.com/doifinder/10.1038/nature24270

[SSS$^+$17b] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[SSSS16] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving," October 2016. [Online]. Available: http://arxiv.org/abs/1610.03295

[Sut91] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[TT15] M. Toussaint and M. Toussaint, "Hierarchical Monte-Carlo Planning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15, no. January. AAAI Press, 2015, pp. 3613–3619. [Online]. Available: http://dl.acm.org/citation.cfm?id=2888116.2888218

[UGA+15] S. Ulbrich, S. Grossjohann, C. Appelt, K. Homeier, J. Rieken, and M. Maurer, "Structuring cooperative behavior planning implementations for automated driving," in *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on.* IEEE, 2015, pp. 2159–2165.

[Wat89] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989.

[WD92] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[Win09] A. F. Winfield, "Foraging robots," in *Encyclopedia of complexity and systems science.* Springer, 2009, pp. 3682–3700.

[WKZG12] M. Werling, S. Kammel, J. Ziegler, and L. Gröll, "Optimal trajectories for time-critical street scenarios using discretized terminal manifolds," *The International Journal of Robotics Research*, vol. 31, no. 3, pp. 346–359, 2012.

[WT02] D. H. Wolpert and K. Tumer, "Optimal payoff functions for members of collectives," in *Modeling complexity in economic and social systems.* World Scientific, 2002, pp. 355–369.

[ZJBP08] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione, "Regret minimization in games with incomplete information," in *Advances in neural information processing systems*, 2008, pp. 1729–1736.

[ZS10] J. Ziegler and C. Stiller, "Fast collision checking for intelligent vehicle motion planning," in *Intelligent Vehicles Symposium (IV), 2010 IEEE.* IEEE, 2010, pp. 518–522.