Automated Planning for Pathfinding in Real-Time Strategy Games

by

Munir Hussain Naveed

A thesis submitted to the

University of Huddersfield

in partial fulfilment of the

requirements for the degree of

Doctor of Philosophy

Department of Informatics

University of Huddersfield

February 2012

Abstract

This thesis is focused on the design of a new path planning algorithm to solve path planning problems in dynamic, partially observable and real-time environments such as Real-Time Strategy (RTS) games. The emphasis is put on fast action selection motivating the use of Monte-Carlo planning techniques. Three main contributions are presented in this thesis. The first contribution is a Monte-Carlo planning technique, called MCRT, that performs selective action sampling and limits how many times a particular state-action pair is explored to balance the trade-off between exploration of new actions and exploitation of the current best action. The thesis also presents two variations of MCRT as the second contribution. The first variation of MCRT randomly selects an action as a sample at each state seen during the look-ahead search. The second variation, called MCRT-CAS, performs the selective action sampling using corridors. The third contribution is the design of four real-time path planners that exploit MCRT and its variations to solve path planning problems in real-time. Three of these planners are empirically evaluated using four standard pathfinding benchmarks (and over 1000 instances). Performance of these three planners is compared against two recent rival algorithms (Real-time D*-Lite (RTD) and Local Search Space-Learning Real-Time A* (LSS-LRTA)). These rival algorithms are based on real-time heuristic search. The results show that a variation of MOCART, called MOCART-CAS, performs action selection significantly faster than the rival planners. The fourth planner, called the MG-MOCART planner, is evaluated using a typical Real-Time Strategy game. The MG-MOCART planner can solve the path planning problems with multiple goals. This planner is compared against four rivals:

Upper Confidence bounds applied to Trees (UCT), LSS-LRTA, Real-Time Dynamic Programming (RTDP) and a rapidly-exploring random tree (RRT) planner. The performance is measured using score and planning cost. The results show that the MG-MOCART planner performs better than its rival techniques with respect to score and planning cost.

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Real-Time Strategy (RTS) games are a key part of the ever growing computer game industry. The most popular and the best revenue making commercial titles like HaloWars, WarCraft and StarCraft are some examples of RTS games. In this game genre, the players collect natural resources (e.g. minerals), build new infrastructures (e.g. army barracks, schools and libraries etc) and engage their armies in war to defend (or capture) territories. A typical RTS game can have one or more players. The player can be a human or a computer. Each player has several characters to control (through commands) in a game play. The number of characters belonging to a player can vary from title to title but the common characters are workers, soldiers, tanks and aeroplanes. The workers are used to gather the minerals or to build new buildings. The other characters (i.e soldiers, tanks and aeroplanes) are used to fight with the enemies. A RTS game can also have neutral characters (for example sheep, bugs or fauna etc) which do not belong to any player in the game. In WarCraft, the sheep walk around randomly. These games are played on huge maps which are

not completely visible to the players. Players can uncover the invisible features - e.g. obstacles or natural resources - of the map by moving their characters to unexplored areas. In a RTS game, the movement of the movable characters (from one location to a goal location on the map) is very often required by the players for different reasons (e.g. to explore the map, to collect and store the minerals or to attack the enemy forces). The main challenging issues for a computer player to find a set of actions to move a character from one location to the goal location are: time constraints, partial visibility and the dynamic game world. Path planning must be performed within a constant (and very small) time interval so that the movement of the character looks real-time. The topology of the game map can be changed anytime during the game play (e.g. due to the construction of new buildings). The path planning problem in a RTS game is a complex and challenging problem (due to the dynamic and partially observable game world) for a computer program to solve under tight time constraints. Path planning for fully observable and static environments is a well explored area in Automated Planning (AI Planning) that deals with the finding of a geometrical path between two locations on a map. A map is usually represented in the form of a grid [74]. A grid is a finite rectangular area which consist of square (or rectangular, hexagonal or triangular) cells [74]. A cell is either traversable or untraversable. A cell is also called a tile. The planning agent always occupies exactly one cell and starts from a traversable cell. The path planning problem with a fully observable and static environment is modeled as a graph search problem [73]. The graph search algorithms (e.g breadth first and depth first) are defined in section 1.2 of this chapter. A* [30] is a commonly known algorithm for pathfinding in static and fully observable environments. A* and its variations are discussed in detail in section 2.1 of Chapter

2.

The path planning problem in a real-time partially observable and dynamic environment such as an RTS game is a less explored area in AI planning. The popular path finding algorithms (e.g. A* and D* [64]) are not applicable in these kind of environments due to the tight time constraints and partial observability. Learning Real-Time A* (LRTA) is a suitable algorithm for solving real-time path planning problems with incomplete information. LRTA solves a path planning problem by interleaving planning and plan execution. In each planning episode, the planning agent searches for an action and executes it. These algorithms impose a limit on the amount of planning in an episode. These episodes are repeated until the character reaches the goal location. LRTA and its variations are discussed in detail in section 2.1.4 of Chapter 2. These techniques have some limitations, for example, they are exponential in the dimensions of state space [17]. Monte-Carlo Planning is a kind of AI planning that is suitable to solve planning problems in dynamic and partially observable environments.

This thesis focuses on the design of real-time Monte-Carlo planners to solve path planning problems in partially observable (and dynamic) environments and compares them against state-of-the-art rival techniques. The Monte-Carlo planners are evaluated using the pathfinding benchmarks. The thesis provides a detailed analysis of the strengths and weaknesses of the Monte-Carlo planners.

The rest of this chapter is organised as follows. Automated planning is defined in section 1.1. Section 1.2 defines the searching techniques which are commonly used in AI planning. The scope of AI planning in RTS games is discussed in section 1.3. A detailed discussion on path planning in RTS is given section 1.4. A list of contribution is given in section 1.5. The outline of thesis is given in section 1.6.

1.1 Automated Planning

Automated planning (AI Planning) is an area of Artificial Intelligence that deals with the study of planning (i.e. selection and organisation of actions) computationally to solve a problem [27].

The conceptual model of AI planning is based on a state-transition system [24] [27]. The state transition system has four elements: a finite set of states, a finite set of actions, a finite set of events and a state-transition function. AI planning is concerned with the selection of a sequence of actions to reach the goal state(s). The restricted planning model (or classical model) assumes that the set of events is always empty. In AI planning, the implicit representation of the state space is common; as the explicit enumeration of the states and their transitions is practically not possible for domains where the number of states are exponentially huge. For example, in classical planning, a state of the world is represented by a set of predicates (in a first order language) which are set true or false by actions (or planning operators). A planning operator has three elements: a parameter list (used as an identity of the operator), preconditions (a list of predicates that should be true (false) before this action is executed) and effects (a list of predicates that represent a resulting state from execution of this action). An AI domain model consists of the list of predicates and the planning operators. An AI planning problem contains the specification of the initial state and the goal state(s). A conceptual planning model is shown in Figure 1.1 (adapted from [27]). It has three parts: a planner, a controller and the state transition system. A planner generates a plan for a given problem specification by using the domain model. The plan can be a sequence of actions or a policy function
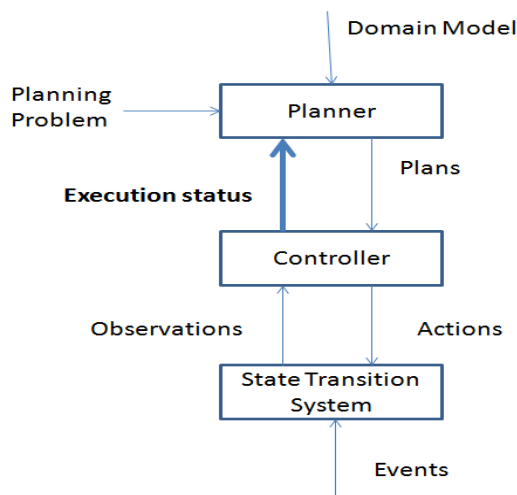
Figure 1.1: A Conceptual Model of AI Planning (adapted from [27]).

(of mapping from states to actions). A controller gets the current state of the system (i.e. observations) from the state transition system and chooses an action according to the plan (generated by the planner). The state transition system evolves according to the actions it receives from the controller. The state transition can also evolve due to an event. The occurrence of an event depends on the internal dynamics on the system (instead of a controller). A controller can send a feedback of the execution status to a planner after an action is executed. This kind of controller is called a closed loop controller. The closed loop controller is used in dynamic planning [27] (e.g. interleaving planning and plan execution). In a conceptual model of planning, a planner is kept logically separated from the domain model. The logical separation of the planner from the domain model gives several advantages, e.g. the planning engines and domain models can be improved, tested and validated independently of each other [62]. Planning Domain Definition Language (PDDL) [48] is a standard language that is used to represent the planning domain model and the planning

problem in AI planning.

A domain world can also be represented by using a simulation model [36] [8]. The use of a simulation model for AI planning is suitable for a domain world which is difficult to model in a planning language (e.g PDDL). Klondike Solitiare [8] and RTS games [3] are examples of such domains.

1.1.1   Deterministic Domain Model

If the application of an action at a state can result in a single next state, then the state transition system is deterministic [27]. The next state is also known as the effect of the action. In a deterministic domain model, every action has only one effect.

1.1.2   Non-Deterministic Domain Model

In a non-deterministic state transition system, application of an action at a state can lead to different possible states [27]. In a non-deterministic domain model, the effect of an action at a state can be a transition to any state from a list of states S_{next} which are possible to reach with the application of that action. The size of this list is always greater than one.

1.1.3   Probabilistic Domain Model

The probabilistic domain model is similar to the non-deterministic except that the transition to the next state is associated with a probability distribution [14]. The transition to a next state depends on a probability distribution such that a state $s' \in S_{next}$ has a greater chance of being the next state if it has a higher transition

probability than the other states in S_{next}. A probabilistic action model with a uniform transition probability model is equivalent to a non-deterministic model.

1.1.4   Sensor Feedback

As shown in Figure 1.1, a controller can get feedback (i.e. observations) from the environment after an action (selected by the controller) is executed. There are three kinds of feedback that a controller can sense. These are Complete, Partial and Null. A state transition system with complete sensor feedback is called a fully observable environment. In a partially observable environment, the controller has a partial sensor feedback.

In a complete sensor feedback, the controller has full information about the current state of the system. With a partial sensor feedback, the controller does not know the current state of the system exactly due to the possibility of the occurrence of different states in the same observation. Partial information from the observation is used to find out the true state of the system at the moment. Null feedback sensor is a kind of blind partial sensor where the controller does not get any information about the observation.

Markov Decision Processes (MDP) is a model that is used to represent a non-deterministic (or probabilistic) domain model under the assumption of complete sensor feedback [27]. This model assumes that the effect of an action is unpredictable. A plan in this model is a policy function.

A non-deterministic domain model with a partial sensor feedback controller can be modeled as a Partially Observable Markov Decision Processes (POMDP) [27].

POMDP maintains two additional elements to the elements of MDP. These additional elements of the model are an initial belief state and a set of observations with an associated probability distribution of getting an observation (that is a member of the set) due to the application of an action at a state of the state space. The belief states are the probability distribution over the real states. This model assumes that the effect of an action on a belief state is predictable. So, a plan in a POMDP is a mapping function (i.e. a policy) from belief states to actions.

A controller without a feedback computes the plan offline and actions are executed without any concern about the feedback from the environment. This kind of controller is called an open-loop controller.

1.1.5 Satisficing Versus Optimal Planning

Each action that a planner chooses to solve the given planning problem incurs some cost. The cost of solution is the sum of the cost of all actions a planner chooses to solve the problem. A planner that generates solutions of minimum cost is called an optimal planner. A planner is satisficing if it solves the planning problems under the given temporal and resource constraints.

1.2 Search Methods

In AI, search is a fundamental part of the problem solving mechanism. The planning engine uses a searching algorithm to find a sequence of actions to solve a planning problem. The searching algorithm searches through the state space to find the solution. The search can be uninformed or informed [57]. An informed search algorithm

traverses through those states of the state space which look promising according to some information or estimate (heuristic). Informed search is also called a heuristic search. An uninformed searching algorithm uses only information that is given in the problem definition. To traverse through the state space, the search algorithm generates a search tree from the initial state by expanding the states seen by the search algorithm. In this section, we discusses those search techniques that use an explicit search tree. The search tree can be in the form of a graph if the same state in the search tree is reachable from more than one path. However, we define these techniques by assuming that one state is reachable from only one path. The search strategies differ with respect to the way they expand the search tree from the current state. A state s (element of the state space) is expanded by finding the next state of each action applicable at s. This set of next states is called the successor states of s. The fringe of the search tree is the collection of the successor states in a search tree that have not been expanded. These are also the leaf nodes of the search tree. A state s' is called a parent state of s if s is generated by s'.

1.2.1   Uninformed Search Strategies

Breadth first search expands the root state first and then its successors and then the successors of the successor states. In other words, Breadth first search expands all the states at a given depth before expanding any state in the next level [57]. This process continues until there is no state left to expand or the goal state is reached. If d is the depth of the search tree and b is the breadth of the search tree then the space and time complexity of this strategy is $O(b^d)$.

Depth first search strategy expands the deepest state in the fringe of a search tree. It picks a neighbouring state of the current state and expands it to next level [57]. In other words, it expands the child states before expanding the siblings states. It keeps expanding the child states unless the deepest most state has no child state. If a state has no successor (child) state, then it is removed from the memory. When the leaf node is removed from the tree, then search backs up to the upper level of the tree and expands an unexplored successor state of the parent state of the removed state. This process continues until the goal state is reached or all successor states of the initial state are expanded. The space complexity of depth first is $O(bd)$.

1.2.2 Informed Search Strategies

An informed search strategy applies problem specific knowledge (other than the definition of the problem itself) to guide the expansion of the search tree to solve a problem using an evaluation (e.g. heuristic or estimate of desirability). Best-first search is an informed search strategy that uses an evaluation function (e.g. heuristic function) to guide the expansion of the look-ahead search [57]. A greedy best first search expands the states which seems closest to the goal state according to a heuristic function. A* [30] is a kind of best-first search that avoids expanding the nodes which appear expensive. The details of A* are discussed in Chapter 2 (section 2.1).

1.2.3 Local Search

Local search algorithms keep the search in the neighbourhood of the current state of the world and move to a neighbouring state that looks promising. Local search

algorithms are not only used to solve the problem of finding a goal from a state but they are also used to solve optimisation problems.

Hill-Climbing is a kind of local search algorithm that searches for the best neighbouring state (of the current state) and moves to the best state [57]. The best neighbouring state is a state in the vicinity of the current state such that it has a higher value (according to an objective function) than the value of the current state and the highest among the neighbouring states of the current state. It stops searching if there is no best neighbouring state at the current state. Hill-Climbing is a greedy local search algorithm. Hill-climbing can get stuck in local maxima if the state – at which it terminates the search – has the highest value with respect to the current neighbourhood of the state but it is not the highest value globally. Simulated-Annealing [37] is a kind of local search that allows the selection of a neighbouring state that has a lower value than the current state to escape the local maxima.

1.2.4   Search Techniques for Monte-Carlo Planning

Monte-Carlo planning techniques use a simulation model of the planning domain to generate a look-ahead search tree [68]. A simulation model uses a stochastic state transition function (to estimate the next state of a state-action pair) and a sampling technique to draw the action samples at each state seen in the look-ahead search. There are two main kinds of sampling techniques: uniform and selective. A uniform sampling scheme selects all actions applicable at a state seen during the look-ahead search. The look-ahead tree expands in a sparse fashion. In a selective sampling scheme, only one action is sampled at a state seen during the look-ahead search in a

simulation. The simulation model also requires a reward function to assign a reward value to each state-action pair seen during the look-ahead search. These rewards are used to estimate the value of each action at the current state. A Monte-Carlo planner chooses an action at the current state which gets the maximum value from the simulation model. Monte-Carlo planning is suitable for solving planning problems in large scale state spaces [19][22].

1.2.4.1 Monte-Carlo Policy Rollout Search

In a Monte-Carlo policy rollout search, a look-ahead search tree is always generated from the current state and is extended either using the uniform or the selective sampling technique. The Monte-Carlo policy rollout with selective action sampling is linear in the depth of the look-ahead search tree. Therefore, it is considered suitable to speed up search in a Monte-Carlo simulation model. These techniques are discussed in section 2.2.3 of Chapter 2.

1.2.4.2 Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) [21] has two main parts: (1) Tree expansion (2) Monte-Carlo simulations (e.g. policy rollout). The look-ahead search tree is expanded using the outcomes of the Monte-Carlo simulations. At the root node, MCTS recursively chooses the best child nodes (according to the action value) to traverse to the leaf node of the look-ahead tree. If the leaf node is not a terminal node, then simulations are run at the leaf node. The outcome of the simulations is used to add another node in the look-ahead tree. This outcome is also back propagated to the root node and updates the value of all actions seen in this episode of traversing. This

process is repeated for a fixed number of times. A detailed discussed of algorithms (e.g. Upper Confidence bounds applied to Trees (UCT) [39]) using MCTS is described in section 2.2.3.

1.3   AI Planning for RTS Games

RTS games provide a challenging and complex environment for the AI player to solve planning problems. Some of the challenging problems for the AI player in RTS games are adversarial planning, path planning, tactical assault planning and resource management [19]. An adversarial planning problem is about making decisions by predicting the response of the enemy players. For example, to find minerals on a map using adversarial planning will involve the exploration of unseen areas which are not in the control of enemy forces. Tactical assault planning is about making decisions to attack the enemy forces. Resource management involves the exploration of the game world to collect resources (e.g. minerals) and deciding about the use of these resources. The most common and popular field of AI to solve these planning problems in RTS games is AI planning. For example, Sailer et al. [59] used Monte-Carlo planning to solve the adversarial planning problem in a RTS game. The simulation model in their work requires information (e.g. position and velocity) about the characters belonging to the opponent players. Balla and Fern [3] applied a Monte-Carlo planner (based on Monte-Carlo tree search) to solve the tactical assault problem in a RTS game called Wargus. Chan et al. [20] applied a classical planning approach in Wargus to solve the resource collection problem. The authors represent Wargus in PDDL with a deterministic action model. Their work applies a sequential planner based on

Mean-ends analysis [53] [26] to generate a plan to reach the goal state. A goal in this planning problem is to reach a certain amount of resources e.g. ten thousand minerals. Mean-ends analysis is a kind of informed search that prefers actions which reduce the difference between the current state and the goal state. The authors represent domain using a deterministic domain model. The results show that the plans generated by the AI planner are comparable to the strategies made by a human-expert.

The exploration of suitable AI planning techniques for RTS games is also useful for other domains, for example in simulated battles in military training tools.

1.4 Path Planning Problem in RTS Games

Each character in a RTS game is supplied with incomplete information about the game world. The line of sight of a character is limited to a small distance (which takes into account occlusion by obstacles blocking places it would normally see). The locations beyond this distance are invisible for the character. The limited view of a character in a RTS game is also known as fog of war. For example, in Warcraft and Starcraft, the concept of fog of war is used to keep the map features (e.g. resources, obstacles and even enemy units etc) hidden from the characters. These features are revealed to a character if the features are in the line of sight of the character.

A path planner is required by a computer player to move a character from one location to another in a RTS game. The character that requires path planning to move to another location is called a planning agent. In this thesis, we address the path planning problem in RTS games as a single agent planning where each character generates its own path plan to solve a planning problem.

The planning agent is equipped with very limited information about the game world. The planning agent has a sensor that can provide local information in the current neighbourhood of the planning agent. This sensor information contains the current position and velocity of the planning agent, the position of the goal state(s), and the set of locations (and their status) that are within the line of the sight of the planning agent. The status of a neighbouring location can be either occupied or empty. If a location is passable, it is called empty, otherwise it is occupied. The planning agent also knows about about the size of the world and the set of actions it can perform. The topology of the world can change during the problem solving. There are two kinds of obstacles in the domain world: static obstacles and dynamic obstacles. The static obstacles stay in one place for the whole game while the dynamic obstacles are the movable characters in the game. The presence of dynamic obstacles make the action model non-deterministic due to the collisions. For example, if a planner chooses an action at location x to move the planning agent to location y on a game map, it is possible that the agent does not move to y when the selected action is executed because y was occupied by another randomly moving object before the selected action is executed.

The main challenging issues for a path planner in this kind of setting are the incomplete information about the domain world, hard real-time constraints, non-deterministic action model and the dynamic changes in the domain world.

1.4.1 Problem Definition

The thesis addresses the single-agent path planning problem in a RTS gaming environment where the topology of the game world changes during the problem solving process. The environment is partially observable i.e. the planning agent can see only a small part of the topology of the game world at any time during the problem solving process. The thesis addresses the path planning problem by assuming the following properties of the game world.

1. The game world has both static and dynamic obstacles. A dynamic obstacle moves randomly and can stop moving for any length of time during the problem solving process.

2. The players do not engage in a war during the game play.

3. The environment is a hard real-time system where a planning agent is bound to respond within a fixed time-limit. The time-limit is set by the environment and usually specified by using the frame rate e.g. in an experimental game (called RC-RTS), the limit is eight frame per second.

4. A planning problem can have one or more goals.

5. The game is run for a fixed number of frames.

1.4.2 Research Objectives

The main goal of this research work is to design new path planning algorithms with an aim to improve the fast action selection mechanism in a RTS game like environment.

The thesis explores new Monte-Carlo planning techniques to achieve fast action selection in a real-time path planner which interleaves planning and plan execution. The new Monte-Carlo planning techniques approximate an answer for a given planning problem by running simulations within predefined tight time constraints. The Monte-Carlo planning techniques, that we present as a contribution, perform selective action sampling in a simulation to generate a look-ahead search tree of fixed depth. The action values are approximated by using a novel reward function which is based on the shortest distance estimate and avoids collision with static obstacles. The sampling scheme also uses a new convergence definition which puts a limit on the number of times an action is sampled in the simulations.

1.4.3 Motivation

Path planning has been extensively explored in computer science and robotics but the majority of the work has been done in fully-observable and static environments. There has been a little improvement in the direction of solving path planning problems in dynamic, incomplete information and real-time domain worlds. The most recent path planners that are suitable for this kind of planning problem are based on two popular pathfinding algorithms known as A* and D*. These algorithms are discussed in detail in section 2.1 in Chapter 2. The problem with these techniques is that they are exponential in the dimensions of the state space. Monte-Carlo planning techniques are one of the suitable AI planning approaches to solve planning problems in high dimensional search spaces [18][39][3]. Another motivation for exploring Monte-Carlo planning techniques to solve path planning is that these techniques are equally

applicable in both holonomic and non-holonomic path planning problems [45]. A holonomic path planning problem is like a man walking in a park i.e the motion of the man is not restricted by his walking speed and the person can move in any direction instantaneously. This is like path planning for a soldier or a worker in a RTS game. The non-holonomic path planning problem is a kind of path planning where the motion is restricted by the velocity of the planning agent e.g. a car's motion. To move a car to a neighbouring location depends on its speed and the steering angle. The main objective of this thesis is to investigate Monte-Carlo planning techniques in solving path planning problems in a dynamic, incomplete information and real-time domain world such as a RTS game. In this sort of environment, time required to search (for an action) is a crucial factor for an AI planner. In a RTS game, a path planner should be capable of finding a satisfactory partial solution as quickly as possible. The main emphasis of this research is put on fast action selection to find a partial solution (within a constant and short time interval) and thus motivating the use of Monte Carlo techniques. The thesis explores these techniques in a RTS game to solve path planning problems. The Monte-Carlo path planning techniques, that we present as a contribution, are not only applicable for solving path planning problems in RTS games but these algorithms are also applicable in other domain worlds which are represented as grids. These techniques are particularly suitable to solve path planning problems in the domain of Military Simulations [32]. The techniques can also be extended to solve path planning problems in the domain of RaceTrack [5]. The main contributions of the thesis are given in section 1.5.

1.5 Contributions

The thesis presents four main contributions which are described the following list.

1. The first contribution is the design of a Monte-Carlo planning technique (called MCRT [52]) that is suitable for solving path planning problems in a dynamic, incomplete information domain (like RTS games) under hard real-time constraints. MCRT performs Monte-Carlo simulations within a fixed time interval. This time interval is independent of the size of the state space and the length of the planning problem. The Monte-Carlo policy rollout simulations use a reward function that combines two domain-independent (but problem specific) heuristics: the shortest distance heuristic and the collision estimate.

2. The thesis also presents two variations of MCRT – MCRT-RAS and MCRT-CAS [51] – as the second contribution. These variations speed up the look-ahead search in MCRT.

3. The thesis presents four real-time Monte-Carlo path planners as the third contribution. These planners interleave planning and plan execution. Three of these planners are designed to solve single journey path planning problems. A single journey path planning problem has one start location and one goal location. These planners are MOCART, MOCART-RAS and MOCART-CAS. The MOCART planner exploits the MCRT technique while the MOCART-RAS and MOCART-CAS planners exploit MCRT-RAS and MCRT-CAS techniques respectively. These planners and their rival techniques are evaluated using four pathfinding benchmarks. The results highlight the strengths and weaknesses of

each planner on these benchmarks. The fourth planner – called MG-MOCART – is designed to solve multiple journey path planning problems. A multiple journey path planning problem has one start location and multiple goal locations. The MG-MOCART planner is evaluated using a RTS game.

4. The thesis also presents the application of UCT and RRT [50] in a typical RTS games (for solving the multiple path planning problems) as the fourth contribution.

1.6   Thesis Structure

The rest of thesis is organised as follows. Chapter 2 describes the commonly used path planning algorithms in computer games based on A* in section 2.1. Some of the variations of A* (for example D*, D* Lite) are suitable for path planning in a dynamic and incomplete information game world but these variations are not real-time. The real-time variations of A* and D* like Learning Real-Time A* (LRTA) [43] and Real-time D*-Lite (RTD) [11] are discussed in section 2.1.4. A recent variation of LRTA called Local Search Space-LRTA (LSS-LRTA) and RTD are selected as suitable rivals for comparison against the planners we designed using MCRT and its variations. The non-deterministic and probabilistic planning techniques are presented in section 2.2. The Monte-Carlo policy rollout techniques are described and reviewed in section 2.2.3. Popular techniques in this area are Upper Confidence Bounds applied to Tree (UCT) and sparse sampling. A non-deterministic version of LRTA, called Real-time Dynamic Programming (RTDP) [5], is described and reviewed in section 2.2.2.1. A Monte-Carlo planning technique, called Rapidly-exploring Random Tree (RRT), is

described in section 2.2.3.1. RRT has been explored extensively in robotics to solve non-holonomic path planning problems.

Chapter 3 provides details of the Monte-Carlo planning techniques (MCRT and its two variations). This chapter describes the design of a rollout in MCRT. The algorithmic details and complexity analysis of MCRT and its variations are described in detail. Two variations of MCRT are described in section 3.5 of Chapter 3. The convergence of action values is defined in section 3.6 of Chapter 3.

Chapter 4 provides details of the four planners that are based on MCRT and its variations. The planners interleave planning and plan execution. This chapter also describes the cases where these planners always reach the goal state (i.e. completeness of the planners). Three of these planners solve path planning problems where the problem has one start location and one goal location (i.e. single journey path planning problems). Details of these planners are given in section 4.1.

The fourth planner – called MG-MOCART - has been designed to solve a path planning problem that can have more than one goal location (i.e. multiple journey path planning problems). The chapter also highlights the situations where a path planning problem can have more than one goal. The details of this planner are given in section 4.2.

Chapter 5 describes the details of the experimental setup for the empirical evaluation of the four planners that are designed to use MCRT and its variations. The MG-MOCART planner (that is designed to solve planning problems with multiple

goals) is evaluated in a real-time strategy game called RC-RTS. The details of the experimental set up for the evaluation of this planner are given in section 5.2. This planner is compared against the closest rivals which are RRT, UCT and RTDP. The other three planners are evaluated using four benchmark maps. These benchmark maps are Arena2, Orz103d, Orz702d and Orz900d. These three planners are compared against two close rivals (LSS-LRTA and RTD).

Chapter 6 provides a summary of the work presented in this thesis. The chapter also highlights the strength and weakness of the main contributions of this thesis. The chapter then describes future work with respect to the possible extensions of MCRT (and its variations) and the application of the four planners (that we present as the contribution of the thesis) in non-deterministic domains e.g. Racetrack.

Chapter 2

Related Research

This chapter describes popular pathfinding algorithms and their suitability to solve the path planning problem in RTS games (the problem has been defined in section 1.4 of Chapter 1). Most of the pathfinding algorithms are based on a best-first search algorithm called A* [30]. We discuss A* and its variations in section 2.1. The variations of A* that are suitable for real-time path planning are discussed in section 2.1.4. This section also highlights the limitations of the variations of A* with respect to their application in a RTS game to solve the path planning problem. Section 2.2 describes the planning algorithms for the domain worlds represented with the non-deterministic or probabilistic action models. The Monte-Carlo planning algorithms are discussed and analysed in section 2.2.3. Section 2.3 summarises the chapter by briefly describing the advantages and disadvantages of the planning algorithms presented in this chapter.

2.1 A* Pathfinding Algorithm

A* is the most commonly known search algorithm that expands look-ahead search in the best-first order to find a path between the initial state and the goal state. It uses the sum of an admissible heuristic value (say h) and a path cost value (g) to guide the look-ahead search. The path cost value (g) of a current state represents the cost of moving from the initial state to the current state. This sum is called an evaluation value (f) of a given state. A* expands the look-ahead search tree using the evaluation value of the states seen in the look-ahead search. The look-ahead search starts from the initial state by finding all walkable neighbours (of the initial state) and selects a neighbour with the least evaluation value (f) to expand. This process continues until it reaches the goal state. Each node generated in the look-ahead search is stored in either of two lists: the OPEN list and the CLOSED list. If a node is seen for the first time, then it is added to the OPEN list. A node in the OPEN list with the least f value is selected to expand. After a node is expanded, it is removed from the OPEN list and added to the CLOSED list. The nodes in the CLOSED list are never re-expanded. In other words, A* expands one node only once. A* stops the search for a solution if the goal state is seen in the CLOSED list or if the OPEN list becomes empty. A* always find the shortest path between two states if the heuristic function is admissible. With a non-admissible heuristic A* expands a smaller number of nodes in the look-ahead search when compared to the number of nodes expanded with an admissible heuristic. But A* with a non-admissible heuristic does not produce optimal solutions. A* is not suitable for dynamically changing domain worlds because it never updates the evaluation value of the states (seen in

the look-ahead search) once it finds a solution for the given problem.

2.1.1   A* Variations

Several variations of A* have been proposed to either speed up A* or reduce the space complexity of A*. The variations that speed up A* either use an abstraction of the state-space (e.g. a game map) or reduce the size of the state space or create a true distance heuristic in a pre-processing phase. The variations of A* that build an abstraction of a game map in the pre-processing phase are presented in section 2.1.1.1

To reduce the space complexity of A*, Iterative Deepening A* (IDA*) [42] expands a node in a depth-first style and without maintaining the OPEN and CLOSED lists. IDA* and its variations are discussed in section 2.1.2.

Dynamic D* is a variation of A* that has been designed to solve path planning problems in a dynamically changing domain world. This kind of search is also known as incremental search. D* and its variations are described in section 2.1.3. The variations of A* and D* that are suitable for path planning in RTS games are discussed in section 2.1.4.

Anytime variations of A* use a non-admissible heuristic (then it is also inconsistent) to find a quick solution and then improve that solution if more time is given to the algorithms. Anytime A* [29] finds a solution faster than A* by using an inadmissible heuristic function and then iteratively improves this solution until the open list is empty. Even Anytime A* is faster in terms of finding the first solution (than A*) but its convergence can take a long time because a node can be expanded many

times (due to the inconsistent heuristic). Anytime Repair A* (ARA*) [47] extends Anytime A* by imposing a limit on the number of times a node is expanded in the look-ahead search. Anytime A* and its variations are not suitable for path planning in RTS games for two main reasons. First, these variations require complete information about the domain world to find the first solution for a path planning problem. Second, these algorithms do not provide a guarantee for the real-time constraints. A guarantee means the algorithm should find a solution under a fixed time interval (for all planning problems) such that this time interval is independent of the size of the state space. Real-time heuristic search (discussed in section 2.1.4) provides such a guarantee and therefore, the real-time heuristic search algorithms are suitable for path planning in RTS games.

2.1.1.1   Abstraction of State Space

The variations of A* that use an abstraction of the game map include Hierarchical Pathfinding A* (HPA*) [15], Partial-Refinement A* (PRA*) [67] and memory efficient abstraction [66]. These variations are faster than the original A* but they produce suboptimal solutions. The quality of the solution depends on the kind of abstraction built during the pre-processing of the map. HPA* divides the map into large rectangular blocks. Each block has some entry points on each border. A block is connected to another block through these entry points. The paths between all entry points of a block are computed and stored in a look-up table during the pre-processing phase. The abstraction and the look-up table are stored in memory for online planning. To find a path between two locations on the original map , HPA* finds the corresponding blocks of the initial and the goal locations. A high level path

between the block of the initial location to the corresponding abstract block of the goal location is computed using A*. The pre-computed paths in the look-up tables are used to refine the abstract path. The refined path is improved by using a post-processing path smoothing phase. The problem with this abstraction is that it allows the path between two blocks through entry points only.

Partial refinement A* (PRA*) builds a multi-level hierarchy of a map by grouping together several locations (connected with each other) into one abstract state. PRA* finds an abstract path at an arbitrary level and refines it within a predefined window (called a corridor) of the low level hierarchies. PRA* finds a partial path and executes it (i.e. interleaving planning and execution). For large maps, PRA* is expensive in terms of memory. A memory efficient abstraction called Minimal-Memory (MM) abstraction [66] combines HPA* and PRA* in such a way that the map is first divided into large blocks and then each block is abstracted by PRA*. This abstraction technique uses less memory than HPA* and PRA* but it is also suboptimal as it finds a path from the centre of one block to the centre of another.

The use of abstraction of a game map for pathfinding has also been explored by using the navigational-mesh [28] and points of visibility (or way-points) [55]. In navigational-mesh, a set of convex polygons is constructed offline such that each convex polygon (e.g. a regular pentagon) represents an obstacle-free (walkable) surface of a game world. The main advantage of this approach is that it is equally suitable for 2D and 3D game maps. As walkable surfaces are computed offline in a preprocessing phase, therefore, a navigational-mesh approach is not applicable for dynamic environments.

In the points of visibility approach, a set of nodes is created offline in such a way

that the path between two nodes is a straight line. These nodes are usually placed on the actual map manually, commonly on the corners of static obstacles. The effectiveness of this approach hugely depends on the topology of the game map e.g. if static obstacles are irregular shapes then this approach requires a large set of nodes and motion can also look non-realistic as characters would not be able to move to many obstacle-free spaces that are not part of the straight line. Another problem with visibility points is that path planning becomes slow if there are a large number of points.

In RTS games, the building of abstractions in a pre-processing phase is not possible because the maps are initially unknown to the players. Therefore, HPA*, PRA*, MM abstraction, way-points and navigational-mesh are not suitable for the path planning problem in RTS games.

2.1.2 Iterative-Deepening A*

Iterative-Deepening A* (IDA*) [42] expands the look-ahead search using an informed depth-first search. It is memory efficient because it does not use the OPEN and CLOSED lists to find the path. It keeps a threshold value to guide the look-ahead search. The main drawback of IDA* is that it can get stuck in cycles in a graph search because depth-first search does not store the states it sees in the look-ahead search.

A variation of IDA* - called Fringe Search (FS) [10] - handles the problem of cycles in IDA*. FS stores the leaf nodes of the current iteration for use in the next iteration. These leaf nodes are used as the starting point in the new iteration. Experiments have

been performed (by the authors of [10]) on 120 maps (extracted from a commercial game called *Baldur's Gate II*). The results show that Fringe Search finds solutions faster than A* and IDA*. All three algorithms find paths of the same quality (i.e. path length).

2.1.3   Incremental Search

Incremental search is a kind of search algorithm that has been designed to solve the shortest path planning problems in dynamic domain worlds. These algorithms reuse the searching efforts done in the previous search and update the edge costs to adapt to the new changes. In this section, we will discuss informed incremental search algorithms that have been proposed to solve path planning problems.

D* [64] (Dynamic A*) is designed for solving path planning in partially known and dynamic environments. It is a kind of A* that dynamically updates the cost of edges if it finds a change in the environment during the search for the solution. D* maintains an OPEN list only. The expansion of a state in the OPEN list depends on a key function. The key function determines the value (or minimum cost) of each state it sees in the look-ahead search. The states in the OPEN list are sorted according to values returned by the key function. The state with the least value is expanded. In D*, the value of a state can increase or decrease due to a dynamic change in the environment during the problem solving process. Therefore, D* finds optimal solutions in the dynamic and initially unknown domain worlds. However, the main drawback of D* is that it does not impose a limit on the number of times a node is expanded in D* and the expansion of a node depends on the change in the value of

the state. D* expands even those nodes that are not useful to explore for the current planning problem. Focussed D* (FD*) [65] solves this problem by using a heuristic function along with the key function to decide what states in the OPEN list should be expanded. FD* has been tested on grids of different sizes ranging from 10^4 states to 10^6 states. Focussed D* has been compared against D* and the results show that FD* solves online planning problems significantly faster than D*.

D* Lite [40] is also an informed incremental heuristic search algorithm which searches for a path between the current location of the planning agent and the goal location using backward search. D*-Lite starts the look-ahead from the goal state towards the current state of the planning agent. In D*-Lite, the g-value of any state s in the look-ahead search is the distance from the goal to s. D*-Lite replans the path if there is a change in the topology of the domain world. However, neither D* nor D*-Lite can work under real-time constraints [41]. A variation of D*-Lite for real-time path planning is presented by Bond et al. [11]. This variation, called Real-time D*-Lite (RTD), is described in detail in section 2.1.4.

2.1.4   Real-Time Heuristic Search

Real-time heuristic search interleaves search and action execution. In each searching episode, the look-ahead search is expanded to a fixed depth in the neighbourhood of the current state of the domain world to find a best neighbouring state that has the minimum heuristic value. An action is selected (at the current state) to move to the best neighbouring state. This action is executed and the domain world changes from the current state to a new state. At the new state, the algorithm again searches

for a promising action (applicable at the new state) using the local information of the domain world and executes the action. The searching algorithm also updates the heuristic values of the current state before it executes the action at the current state. The real-time search algorithm repeats the cycle of searching for a promising action at the current state, updating the heuristic value of the current state and executing the action. This cycle is repeated until the goal state is reached. Learning Real-Time A* (LRTA) [43] is an example of real-time heuristic search.

LRTA uses a breadth-first search to find all successor states of a current state in a look-ahead search of depth one. Then it selects a best successor state. A best successor state is a neighbouring state that has the minimum cost as compared to other successor states in the look-ahead search. A cost is the sum of two distances; a distance from current state to the successor state and an estimated distance (i.e. heuristic value) from the successor state to the goal state. The heuristic value of the current state is modified and the action is executing leading to the selected successor state. The same process continues until the goal state or a stopping condition is reached.

During the search episode, the heuristic value of the current state is modified if this heuristic value is smaller than that of the best successor state. This modified heuristic value is stored in memory for future use. This modification, or tuning of the heuristic values, is called the learning of the heuristic function or the learning mechanism of LRTA. LRTA suffers from two main problems:

(1) It suffers from the heuristic depression problem [34]. A heuristic depression is a region of the state space where each state in that region has a smaller (or equal) heuristic value than the heuristic value of the states surrounding that region. In other

words, a heuristic depression is a set of connected states with inaccurate heuristic values.

(2) Learning of the heuristic is slow because it only updates the heuristic of one state in a searching episode.

To solve the first problem, Ishida [34] proposes the use of A* in a LRTA search to get planning search out of the heuristic depression.

Search and Learning Algorithm (SLA) [61] is a variation of LRTA that updates the heuristic value of two states – the current state and the previous state – in a planning episode. It backtracks to update the heuristic value of the previous state. SLA improves the learning speed compared with LRTA but it is slower than LRTA in terms of time of search per episode (even with look-ahead search of depth one) because it requires time to backtrack before executing an action.

Weighted LRTA [60] uses an inadmissible heuristic function to speed up convergence of the learning process and explores a smaller number of states than LRTA with an admissible heuristic function. However, the inadmissible heuristic increases the suboptimality of the solutions.

Learning Real-Time Search (LRTS) [16] is a variation of LRTA that performs a look-ahead search of depth greater than one. LRTS uses an inadmissible heuristic function to speed up the learning process. LRTS uses a max-min rule to update the heuristic value of the current state. According to this rule, if the heuristic value of the current state is higher than the best successor state then the heuristic value of the current state is modified to the heuristic value of the successor state which has the highest heuristic value among all the successor states. Another notable feature of LRTS is the backtracking to the previous state. However, LRTS backtracks at a state if the

47

change in the heuristic value of that state exceeds a user specified threshold value. The worst case time complexity of LRTS is exponential in the dimensionality of the state-space.

Local Search Space LRTA (LSS-LRTA) [41] is a variation of LRTA that uses A* to build the look-ahead search of fixed depth (greater than one). It uses Dijkstra's Algorithm to update the heuristic value of all states which are seen in the look-ahead search. LSS-LRTA accelerates the learning process by updating the heuristic value of all states present in the look-ahead search in a planning episode. In a planning episode, it can find a partial path of length greater than one but the length of the partial path never exceeds the boundary of the local space (built by A*). Due to this feature, LSS-LRTA requires fewer searching episodes than LRTA (and LRTS) to solve a path planning problem. In LSS-LRTA, A* maintains the OPEN and CLOSED lists. The same lists are used for Dijkstra's algorithm [25], however, Dijkstra's algorithm uses $h(s)$ values to sort the OPEN list and selects the state with minimum $h(s)$ (rather than f values) to propagate the information. Dijkstra's algorithm updates $h(s)$ of the states which are in the CLOSED list and are inconsistent. The experiments are performed on maps from a commercial game called Baldur's Gate II. The results show that LSS-LRTA performs significantly better than D* Lite [40] when performance is measured using the search time and length of the path found by the planner.

Real-Time D* (RTD*) [11] is a variation of real-time heuristic search that combines informed incremental search with real-time heuristic search. In other words, it is a hybrid of local and global search approaches to solve a planning problem. RTD* uses LSS-LRTA (or LRTA) for the forward search and as a local planner. D* Lite (or Anytime D* (AD*) Lite [46]) is used as a global search algorithm in RTD* and

performs backward search from the goal state to the agent's currently occupied state. To share the computational limit between both searching approaches in a planning episode, RTD* uses a parameter called *LocalRatio*. *LocalRatio* multiplied by the computational limit gives the time slot for the local search and the remaining time is used by the global search (i.e. D* Lite or AD* Lite) in a planning episode. The action selection in a planning episode depends on whether the global search reaches the current location of the planning agent or is terminated due to the time limit. If the global search reaches the agent's current location then the action is selected according to the first node of the path found by global search otherwise RTD* uses LSS-LRTA to select an action. RTD* does not hold the property of completeness in a dynamic world. RTD* is empirically evaluated (in a static domain) on three maps of size 161x161 (taken from the game $WarCraft$). To create a dynamic world, the authors of [11] build a domain called *Room Domain* where a grid of 100x100 cells is divided into the rectangular rooms. The dynamic world is also fully observable i.e. the planner has complete information about the domain world. In all planning problems, the start location is fixed at the upper left corner and the goal is fixed at the right bottom corner of the map. The doors between the rooms are closed or opened randomly. Performance is measured using sub-optimality. Sub-optimality of a planning algorithm is calculated by using the ratio of the length of the solution found by the algorithm to the optimal length. RTD* is compared against LSS-LRTA and LRTA*. In the static domain, RTD* uses LRTA as the local search algorithm and D* Lite as the global search algorithm. The results in the static maps show that RTD* with a *LocalRatio* of 75% performs better than LSS-LRTA. In the dynamic and fully observable world, RTD* solves planning problems with significantly smaller

49

sub-optimality than LSS-LRTA.

aLSS-LRTA [31] is a variation of LSS-LRTA that improves LSS-LRTA search to avoid heuristic depressions. aLSS-LRTA redefines the heuristic depressive region in a state space. According to this definition, a state s is in a heuristic depression if there is another state s' outside the heuristic depression such that the heuristic value of s is smaller than the sum of the distance (between s and s') and the heuristic value of s'. If a state s is identified as a part of the heuristic depression, then aLSS-LRTA selects the next state (say s') of s such that s' is not a part of D (heuristic depression) and has the minimum f value among all states (in the look-ahead space) which are not a part of the heuristic depression. aLSS-LRTA is evaluated using six pathfinding benchmarks. The authors selected 300 planning problems from each benchmark [31]. Performance is measured using the average solution length, search time (i.e. total time of search to solve all 300 planning problem on a benchmark map) and time per search episode. aLSS-LRTA is compared against LSS-LRTA. The results show that aLSS-LRTA performs significantly better than LSS-LRTA in solution length and search time.

To avoid re-expanding already explored states in a look-ahead search of real-time path planning, Time-Bounded A* (TBA) [9] uses a combination of A* and real-time heuristic search. TBA finds the solution for a real-time path planning problem by interleaving planning and plan execution. It also keeps the OPEN and CLOSED lists in memory during all planning episodes until the agent reaches the goal state. In each planning episode, TBA expands the look-ahead search (using A*) for a fixed number of nodes. The new states are put into the OPEN list. After the expansion of the nodes, TBA backtracks the best state of the OPEN list to find a path to the

current state or to the start state if the current state is not on the backtrack path. TBA uses the CLOSED list for the task of backtracking. TBA also imposes a limit on backtracking in an episode. It backtracks only a limited number of nodes per action. Backtracking could take more than one planning episode to reach the start node. However, TBA can never reach the start state through backtracking if the path from the current location to the start location is blocked due to a dynamic change in the domain world [11]. Due to this drawback, TBA is not suitable for path planning in dynamic domain worlds.

2.2 Non-Deterministic and Probabilistic Path Planners

In this section, we discuss the planners that have been proposed to solve planning problems in domains with non-deterministic and probabilistic action models. Markov Decision Processes (MDP) is a common representation for a planning problem with a non-deterministic or probabilistic action model [27][14]. Popular planning algorithms used to solve MDPs are described in section 2.2.1. There are two kinds of planning algorithms that are used to solve MDPs. The first kind is based on Dynamic Programming and the second are Monte-Carlo planning algorithms. The Monte-Carlo planning algorithms are also very similar to the algorithms in Reinforcement Learning [68]. Monte-Carlo planning algorithms and reinforcement learning algorithms can solve MDP based planning problems without the presence of the internal dynamics (i.e. state transition probabilities). A difference between the the two approaches is

discussed in section 2.2.4. Sampling based path planning algorithms like Rapidly-exploring Random Trees (RRT) are also described in section 2.2.3.1.

2.2.1 Markov Decision Process

A Markov Decision Process model has three main components [57] [27] which are:

1. A finite set of states (S)

2. A finite set of actions (A)

3. State Transition Probability $(P(s, a, s'))$.

$P(s, a, s')$ represents the probability of reaching the successor state $s' \in S$ when the action $a \in A$ is applied in the state s. In a MDP, the state transition function $T(s, a)$ probabilistically selects the successor state $(s' \in S)$ of $s \in S$ for an applicable action $(a \in A)$. The transition probability depends only on the current state and does not depend on the predecessor states. This is called the Markov property.

A solution, for the planning problem represented by a MDP, specifies what action the planning agent should select at any state $s \in S$ where s is the state the planning agent visits. This solution is called a policy (π). π is a mapping from the state to action i.e. $a = \pi(s)$.

There are two functions which are commonly used to compute an optimal policy. These functions are the cost function and the reward function. A cost function $C(s, a)$ represents the cost of taking an action a at s while a reward function $R(s, a)$ represents a reward value of a state transition $T(s, a)$ (in some work (e.g. [57]), the reward function is represented by $R(s)$ i.e. a reward for each state). These two functions give the values for one transition but the optimal policy chooses an action

based on the values computed by considering the future expected cost or reward of an action. This expected reward of action a at s is called an action value $(Q(s,a))$. In [27], $Q(s,a)$ is represented by using the cost function only, while in [58], $Q(s,a)$ is computed by using only the reward function. If $Q(s,a)$ is computed using a reward function, then the optimal policy yields the maximum value of $Q(s,a)$. An action a at state s is the best if $Q(s,a)$ is higher than all other actions applicable at s. $V(s)$ is the state value of s and it is computed using the value of the best action applicable at s (as shown in Equation 2.1).

$$V(s) = \max_a Q(s,a) \tag{2.1}$$

2.2.2 Dynamic Programming

A Dynamic Programming (DP) approach computes $Q(s,a)$ and $V(s)$ using the Bellman equation [6]. A general form of the Bellman equation is shown in Equation 2.2 where $\alpha \in [0, 1]$ is the discount factor. The intuition behind using the discount factor in computing the expected reward is to give more importance to the reward (i.e. $R(s,a)$) of the current state s than the rewards of the future states.

$$Q(s,a) = (R(s,a) + \alpha \sum_{s' \in S} P(s,a,s') \times V(s'))) \tag{2.2}$$

There are two main algorithms that use the Bellman equation to solve a MDP based planning problem. One is Policy Iteration (PI) algorithm and other is Value Iteration (VI) [6]. Both these algorithms search the whole state space to compute $Q(s,a)$ and $V(s)$ for every state s. These algorithms are polynomial in the size of the state space. Both VI and PI perform backward search to update the policy values in an iteration. The main difference between PI and VI is the focus of their search for the optimal

53

solution in the policy space. PI starts from a random policy and improves it in each iteration. VI starts from random state values $(V(s))$ for each state $s \in S$ and updates them in each iteration. Computationally, an iteration of PI is more expensive than that of VI as PI sweeps through the entire state space twice in an iteration whereas VI traverses through the entire state space only once. These algorithms keep searching for the optimal values until there is no improvement either in the policy or in the state value. Another stopping condition is proposed by Williams and Baird [76]. According to this definition, VI or PI are converged relative to a parameter called the Bellman residual or Bellman equation error (θ). If the absolute difference between the values of every state $s \in S$ in two consecutive iteration is smaller than θ, then the dynamic programming algorithm is stopped. This convergence definition guarantees the solution of near-optimal quality.

VI and PI are not suitable for large scale state spaces because these algorithms traverse the whole state space in an iteration. A less expensive kind of dynamic programming that does not traverse the whole state space in an iteration is called Asynchronous Dynamic Programming (ADP) [7]. Real-time Dynamic Programming (RTDP) is an example of ADP. The details of RTDP and its variations are given in section 2.2.2.1.

2.2.2.1 Real-Time Dynamic Programming

Real-Time Dynamic Programming (RTDP) [5] is an asynchronous approximate dynamic programming algorithm. It uses a heuristic function to estimate the initial state values and then applies an asynchronous value iteration to improve these values. It only updates the value of those states which are seen during the look-ahead search. RTDP repeatedly performs a look-ahead search from the current state s. The

look-ahead search is kept focussed towards the goal state in all iterations. In each iteration, it finds actions to reach the goal location. At each state s' in the look-ahead search, it selects an action a that has the highest value. RTDP updates the value of s' using the value of the best action and then finds the next state using the stochastic state transition function. The stochastic transition function uses the probability distribution. In an iteration, the algorithm stops expanding the look-ahead search when it reaches the goal state. The iterations are run until the termination condition is reached. The authors in [5] did not specify any bound on the number of iterations in RTDP to tune the policy function for a given state. However, in the work of [13], the authors use the Bellman residual (error) to set the termination condition in RTDP. In this work ([13]), RTDP stops its iteration if the error is less than ϵ. RTDP convergence (e.g. when the error is smaller than ϵ) can be slow as it always selects the most likely next states in the look-ahead search, and can ignore the potentially useful part of the search space that seems unlikely to be the next states of the best actions.

To speed up RTDP convergence, a variation of RTDP - called Label RTDP (LRTDP) [13] - uses a labeling scheme. According to the labeling approach, a state s is labeled as a solved state if the residual of s is less than ϵ. LRTDP maintains two lists: OPEN and CLOSED. The OPEN list keeps all states that have been seen but not expanded. The CLOSED list stores the states that have been expanded. If any state that is in the CLOSED list has a Bellman residual of less than ϵ it is declared as a solved state. In LRTDP, the root state (initial state of the planning problem) is simulated for a certain number of iterations to declare it as solved. LRTDP is explored and compared against Value Iteration and RTDP in the Racetrack domain.

In Racetrack, a car is moved (by a planner) from its initial position to the goal loca-

tion on a race track. The race track is a grid of cells (x, y) and some of the places on the grid are slippery. If a car is on the slippery part of the track, then the actions may not have their intended effects. The experiments are performed on nine different tracks. The sizes of the tracks range from 9312 to 239089 cells. The experiments are performed using two initial heuristics: $h(s) = 0$ and $h(s) = h_{min}$. h_{min} is the initial estimate of the policy value for each state $s \in S$ and is always greater than 0. The performance is measured in terms of convergence time. The results show that LRTDP converges faster than VI. RTDP is the slowest performing algorithm in these experiments and its convergence time always exceeded the threshold. The authors used ten minutes as a threshold in all experiments. General Planning Tool (GPT) [12] [14] is a planning tool that provides the implementation of both RTDP and LRTDP. LRTDP and RTDP can be applied to solve path planning problems in RTS games if the state transition probabilities are available prior to the start of the game. It is not possible to compute the probability distribution in a pre-processing phase as the game map is initially unknown to all players. A variation of RTDP is designed as a suitable planning algorithm for comparison with our contributions. In this variation, we made two changes in the original RTDP. First, to guarantee a solution within a fixed time, RTDP stops a trial if the look-ahead search reaches a depth d. Second, to arrange a probability distribution for RTDP, we use an online mechanism to build $P(s, a, s')$. The details of this mechanism are given in section 3.2.1.3.

2.2.3 Monte-Carlo Algorithms

Monte-Carlo algorithms are suitable for solving a MDP based planning problem if the state transition probabilities are not available in a domain world. Monte-Carlo algorithms use a simulation model to estimate $Q(s, a)$ for all $a \in A$ at any state $s \in S$. The simulation model generates a sequence of samples (of states or actions) to generate a look-ahead search and then uses a reward function to estimate the action values.

A Monte-Carlo planning algorithm proposed by Tesauro [72] computes the optimal policy to play the game of Backgammon. This technique is called Monte-Carlo policy rollout. Backgammon is a stochastic board game with approximately 10^{20} states. Monte-Carlo policy rollout estimates the action values for the current state of the game by running several Monte-Carlo simulations. At each state of the game, the policy rollout algorithm finds all applicable actions (i.e. possible next moves) and then runs several simulations for each action. The Monte-Carlo policy rollout algorithm, proposed by Tesauro, requires several iterations of the Monte-Carlo simulations per move (at a state of the board game). The simulations are performed in parallel on two supercomputers (IBM SP1 and IBM SP2) to speed up the simulations and to select a move within a fixed time limit. The basic idea behind this Monte-Carlo policy rollout algorithm is to start Monte-Carlo simulations using a random or sub-optimal policy and improve it in each Monte-Carlo simulation. The initial policy is called a base policy. In the start of the Monte-Carlo search, this base policy is used to generate the look-ahead search. But in the subsequent iterations, the recently improved policy is used. The policy is greedy with respect to the action value. It always selects the best

estimated action at any state $s \in S$ in a simulation (as shown in equation (2.3)). A Monte-Carlo simulation is also called a rollout.

$$\pi^{'}(s) = \arg\max_a(Q^{\pi^b}(s,a)) \tag{2.3}$$

In each rollout, the game (started from the current state) is played until its end. If there is a win for the planning agent, the backup value is a positive number for this look-ahead trajectory otherwise it is a negative value. The backup values computed over several rollouts are averaged to compute the action value.

In [72], the authors explore three different kinds of base policies which are: a pure random player, single-layered neural networks and multi-layered neural networks. The improved policies (by Monte-Carlo policy rollout) are tested against a Temporal Difference learning (TD) based backgammon player, called TD-Gammon 2.1 (details of TD are given in section 2.2.4). Performance is measured using the game score. Each improved policy plays 500 games again TD-Gammon 2.1. The results show that the best improved policy is better than TD-Gammon 2.1.

Tesauro's policy rollout algorithm is expensive for solving the MDP based planning problems as it requires several iterations of simulations for each applicable action in a state and the look-ahead search, in a simulation, is expanded until the terminating state is reached.

Kearns et al. [36] introduce a sparse sampling approach to solve a MDP by using Monte-Carlo simulations. Sparse sampling expands the look-ahead search of a fixed depth H and also puts a bound on the number of times an action is sampled (C) in a simulation. At the current state s, for each state-action pair (s, a), it finds the next state s_{next} using the stochastic state-transition function $T(s,a)$. The next state

is expanded in the same style and the process continues until the depth of the tree is H. The value of each sampled state-action pair (s, a) at depth H is estimated using the equation (2.4).

$$Q(s, a) = R(s, a) + \alpha \frac{1}{C} \sum_{i=1}^{C} (V(T(s, a))) \tag{2.4}$$

Kearns et al. [36] define the convergence of a sparse sampling technique relative to a parameter called $\epsilon > 0$. The authors also propose a bound on H ($H = \log_\alpha(\epsilon)(1 - \alpha)/R_{max}$)to guarantee the convergence of the algorithm relative to ϵ. R_{max} is the maximum value of the reward function. A good feature of sparse sampling is that it provides a guarantee for the near-optimal solutions. The difference between the optimal solution and the sparse sampling solution is at most ϵ. The time complexity of sparse sampling technique, per simulation, is $O((|A|.C)^H)$. Therefore, it is also an expensive approach for the MDPs with the large scale action spaces.

Auer et al. [2] present two Monte-Carlo sampling techniques that select only one action (as a sample) per state in a rollout. It selects the best action at each state seen during the look-ahead search. Because this sampling approach is selective, therefore, the approach should balance the trade-off between the use of the best action and the exploration of new actions. The authors of [2] present two methods of balancing the trade-off of exploitation and exploration. These methods are based on the number of times an action has been sampled in the previous Monte-Carlo simulations. The first method is called Upper Confidence Bound 1 (UCB1). UBC1 uses a logarithmic approach to balance the trade-off as shown in equation 2.5. The first term $Q_{av}(s, a)$ in equation 2.5 is the average value of an action a at state s and the second term is the exploration payoff where n is the number of rollouts done so far and n_a is the

number of times a has been sampled at s since the start of the simulations.

$$Q(s,a) = Q_{av}(s,a) + \sqrt{\frac{2\ln(n)}{n_a}} \qquad (2.5)$$

The second sampling technique presented in [2] is called UCB2 which uses an exponential statistical function (as shown in equation (2.6)) to balance the exploration of new actions and exploitation of the already explored best actions. In equation 2.6, $\tau(r)$ is the exponential function $(\tau(r) = (1 + \alpha)^r)$. r is the number of simulations in which an action has been sampled. α is the algorithmic parameter that can have a small value between 0 and 1. In UCB2, an action a (that maximises $Q(s,a)$) is sampled for $(Round(\tau(r+1) - \tau(r)))$ times in a simulation. n is the total number of simulations. UCB2 is more expensive than UCB1 in terms of the number of times an action is selected as a sample in a simulation.

$$Q(s,a) = Q_{av}(s,a) + \sqrt{\frac{(1+\alpha)(\ln(\frac{e \times n}{\tau(r)}))}{2\tau(r)}} \qquad (2.6)$$

In [2], the authors also present a variation of UCB1, called UCB1-Tuned. This variation uses the variance (V_a) of an action a to compute $Q(s,a)$ (equation (2.7)) at the nth simulation. The variance V_a of $a \in A(s)$ at state s is computed using equation (2.8) where n_a is the number of times a is sampled in n simulations. n is total number of simulations. The factor $\frac{1}{4}$ is the maximum variance of a Bernoulli random variable.

$$Q(s,a) = Q_{av}(s,a) + \sqrt{\frac{\ln(n)}{n_a} \times \min\left(\frac{1}{4}, V_a(n_a)\right)} \qquad (2.7)$$

$$V_a(s,n_a) = \left(\frac{1}{n}\sum_{i=1}^{n_a} Q_i(s,a)^2\right) - Q_{av}(s,a)^2 + \sqrt{\frac{2\ln(n)}{n_a}} \qquad (2.8)$$

60

UCB1-Tuned and UCB2 are explored empirically to solve 2-armed and 10-armed bandit problems. Performance is measured using the number of times the optimal arm is pulled. The results are averaged over 100 runs. The results show that UCB1-Tuned performs better than UCB2.

Kocsis and Szepesvári [39] propose a variation of UCB1, called Upper Confidence Bounds applied to Trees (UCT), that performs selective sampling according to equation (2.9).

$$Q(s,a) = Q_{av}(s,a) + C_p\sqrt{\frac{\log(n_s)}{n_{s_a}}} \tag{2.9}$$

C_p is the domain specific parameter such that $C_p > 0$ (it is tuned empirically for a domain). n_s is the number of times the state $s \in S$ has been seen and n_{s_a} is the number of times an action $a \in A(s)$ has been selected as a sample since the start of Monte-Carlo simulations. At the current state of the domain world, UCT extends the look-ahead search by using two policies: UCB1 and Monte-Carlo policy rollout. UCB1 is used to reach the leaf node (of the look-ahead search tree) in a simulation and policy rollout is used to evaluate the leaf node. In [39], UCT is empirically evaluated by using two domains: the sailing domain and P-game tree [63]. The sailing domain is a stochastic domain where a sailboat finds the shortest path between two locations on a grid map. The wind fluctuation causes the stochastic effects of the actions in the domain. UCT is compared with RTDP and a heuristic based sampling approach [54]. Performance is measured using the number of samples required to reduce the error to 0.1. The error is the difference between the expected reward value of the best action chosen by the algorithm and the optimal value. The experiments are run on grids of

sizes ranging from 2×2 to 40×40 cells. The results show that UCT performs far better than RTDP and the heuristic based sampling approach on all grid sizes. In P-game (a board game), UCT is compared against MinMax Alpha-Beta pruning [38]. Performance is measured in terms of failure rate per iterations. The failure rate is the frequency of choosing the wrong move when stopped after a number of iterations. The experiments are performed on two game trees: one with a branching factor $B = 2$ and depth $D = 20$ and the other with $B = 8$ and $D = 8$. The results show that UCT converges to the correct move selection in the order of $B^{D/2}$. UCT and Alpha-Beta show a logarithmic profile, however, UCT converges significantly faster than MinMax Alpha-Beta.

A variation of UCT has been explored in a RTS game by [3] to solve the tactical assault problem. The tactical assault problem involves the movements of the troops to launch an attack on the enemy forces and destroy them. Under the uncertainty and dynamic nature of the domain world, the tactical assault problem in RTS games requires planning of the troops movements in defensive or offensive positions and optimising the parameters like time and health. Optimisation of time means the friendly troops should quickly launch an attack and destroy the enemies. Health optimisation means the planner finds actions that involve no or very small loss of health of the planning agent. The authors use rollout policy iteration with UCT-like action sampling approach to generate plans to solve the tactical assault problem. The UCT action sampling given in equation (2.9) is applicable in a RTS game if the reward values of the actions are in the range [0,1]. For actions having rewards beyond this

range, the authors present an action selection mechanism given in equation (2.10).

$$Q(s, a) = Q(s, a) + \frac{1}{n_{s_a}}[R - Q(s, a)] \tag{2.10}$$

n_{s_a} is the number of times action a is sampled at s, R is the reward of the look-ahead trajectory generated in one rollout. R is computed with respect to an optimisation parameter (time or health). This sampling approach does not have a theoretical background like original UCT has, however, the experimental work highlights the importance of this sampling technique. The authors explored the UCT-variation in Wargus (an open source game for researching RTS games). The experiments are performed on a map of size 128×128 with 12 different game scenarios. Each scenario differs from the others with respect to the number of friendly and enemy units, their groups and their locations on the map. In a scenario, the number of units can range from 10 to 20 per side and groups can be from 2 to 5. The UCT variation is run for optimisation of time and health separately as R can be associated with one parameter at a time. The UCT variation is compared against five base players which are as follows:

(1) Random Player : It selects an action randomly at any state $s \in S$.

(2) Attack-Closest Player: It selects an action to attack the nearest enemy units.

(3) Attack-Weakest: This player launches an attack on enemy units with the lowest effective health.

(4) Stratagus-AI: The default AI of the tool.

(5) An experienced human player.

Performance is compared in terms of time and health. Time is measured using the number of game cycles and health is measured using the number of hitpoints left

at the end of the game. Fewer game cycles and more hitpoints represent better performance. The results show that the UCT-variation (run with 5000 rollouts) performs better than the baseline players in time when R is computed for optimising the time parameter. The UCT-variation achieves higher hitpoints than the other planning techniques if R is computed to optimise the health parameter. However, the UCT-variation performs worse than the other players in time if R is computed for optimising the health parameter. The results show that UCT-variation can optimise only one performance parameter at a time. The authors of [3] also highlight another limitation of this variation of UCT; the simulation model of this UCT-variation is expensive in terms of computing time and is not suitable to work under tight real-time constraints.

2.2.3.1 Sampling-based Path Planners

Rapidly-exploring Random Tree (RRT) [45] is a sampling based path planning approach that has been explored in the area of Robotics movement. It is also a kind of Monte-Carlo search that uses simulations to build look-ahead search incrementally. RRT uses a sequence of random samples (of states) to incrementally build a search tree. However, RRT is not a planner itself. This is a data structure and a random sampling technique. Therefore, it is combined with a planner to solve the path finding problem [45]. For example, RRT-Connect [44] builds two RRTs during the online search: one tree with the start state as the root and the other with the goal state as a root node. A heuristic planner tries to connect both trees and if the trees are connected then a path is returned for execution.

Probabilistic Road Map (PRM) [1][4] and Potential Field (PF) [33] are the sampling

based path planning approaches that have been explored in robotics. PRM builds a graph of the look-ahead search by using random samples. This approach is suitable for static and partially observable environments but in the case of dynamic settings, the look-ahead search graph can become invalid if a dynamic change in the domain world creates a static obstacle at any node of the graph.

PF generates a potential field, starting from the state location of the robot, towards the goal location in such a way that the potential attracts the goal location and repulses the static obstacles. This approach requires complete information about the domain world, therefore, it is not suitable for partially observable environments.

2.2.4   Monte-Carlo Planning Vs Reinforcement Learning

Reinforcement learning techniques (e.g. Temporal Difference TD [68]) can also be used to solve a MDP. The main difference between reinforcement learning (RL) and Monte-Carlo planning techniques is that RL techniques are model free (i.e no state transition model and reward function). RL techniques learn the action values from trial and error interaction with the environment. The difference between Monte-Carlo planning techniques and RL techniques can be expressed as: the former learn action values from the simulated model of the domain world while the latter predict the action values by directly interacting with the domain world.

The basic idea behind RL techniques is to observe the current state of the environment, select an action (which is applicable at the current state), execute it and then obtain a reward (or a punishment) from the domain world. In this section, three popular RL techniques - Q Learning, TD and Dyna - are briefly described.

65

TD [69] uses a back propagation approach [56](a commonly used learning approach in Neural Networks) in an online fashion to estimate the action value. The difference - between the values of an action in two successive time steps - is used as an error for the delta rule. In other words, it computes the estimate of an action value at the current time step and also for a future time step. The difference of these two estimates is used to update the weights of a fully connected neural network where the neural network is used as an estimation function. In TD, the estimation function can be any supervised learning method that is based on the delta rule, (e.g decision tree, table, radial-basis function).

A variation of TD, called TD-gammon [71], is explored in the domain of Backgammon. TD-gammon uses a multi-layered neural network as an estimation function to predict the action values in TD. TD-gammon is trained offline for several episodes and then the trained TD-gammon is played against a human player (ranked 11th best Backgammon player in the world at the time of the match). The results show that TD-gammon (version 2.1) is comparable with the human expert.

Q-learning [75] is also a kind of model-free reinforcement learning that observes the current state s of the domain world, selects an action a with the highest action value, executes it, receives an immediate reward from the domain world and updates $Q(s, a)$. The action value $Q(s, a)$ is updated by using the immediate reward and the best action value of the new state seen by the planning agent after execution of a at s. The main problem with this algorithm is that it can visit the same state infinitely many times before the action values converge.

Q-learning algorithm is also similar to Learning Real-Time A* (LRTA, described in section 2.1.4) as both algorithms interleave planning and plan execution. However,

both algorithms are very different from each other with respect to the estimation of the action values.

Dyna [70] is a model-based reinforcement learning algorithm. It observes the current state, selects an action (applicable) at the current state, executes it, updates action values and builds a model (of the internal dynamics) of the domain world based on the trial and error learning experience.

2.3   Summary

In this chapter, we discussed several path planning algorithms that have been explored in different domains. The complete search algorithms are optimal to solve path planning problems in deterministic domains. However, these algorithms are not suitable for the large-scale state spaces as the searching efforts of these algorithms depend on the size of the state space. Incremental search algorithms (D* and D*-Lite) are also applicable for the initially unknown and the dynamic domain worlds. Anytime Algorithms also reuse the searching efforts done in the past. However, these algorithms cannot work under tight time constraints due to the use of global search for action selection.

The path planning algorithms that use the abstract representation of the game map (e.g. HPA*, PRA*, navigational mesh and way-points etc) require a lot of pre-processing effort and these algorithms are not suitable if the game maps are not completely available before the start of the game. For path planning in RTS games, the complete search (including incremental search algorithms) and anytime planners are not suitable because these algorithms do not guarantee a solution under tight time

constraints. Real-time heuristic search and Monte-Carlo search based path planners are suitable for path planning in RTS games for obvious reasons as RTS games offer an incomplete and dynamic game world to the AI players to solve planning problems under tight time constraints. RTS games also have huge state and action spaces. One of the main advantages of the Monte-Carlo search (e.g. policy rollout) over incremental search and real-time heuristic search is its suitability for the high-dimensional domain worlds.

The state-of-the-art path planning approaches e.g. LSS-LRTA and RTD have been used for solving path planning problems in the context of real-time path planning in partially observable environments. However, these techniques have some limits with respect to fast action selection. For example, LSS-LRTA is exponential in the dimensionality of the state-space. RTD performs a global backward search for the action selection at the current state of the planning agent. RTD is not only expensive in term of the action selection but it is also not useful in a partially observable environment if most of the states surrounding the goal state are occupied by static obstacles.

Recent Monte-Carlo planning techniques (e.g. UCT, RRT and UCB) are suitable for solving planning problems in large state spaces, primarily due to their selective action sampling approaches, but these techniques does not work under tight time constraints. The thesis presents new Monte-Carlo planning techniques which are based on selective action sampling approaches (i.e. UCT and RRT) and work under tight time constraints. The details of these techniques are given in the next chapter.

Chapter 3

Real-Time Monte-Carlo Planning Algorithms

In this chapter, we describe the details of the main contribution of thesis i.e. the Monte-Carlo Real-Time planning technique (called MCRT) to solve real-time path planning problems in the domain of RTS games. The path planning problem has been defined in section 1.4 of Chapter 1. MCRT uses two sampling schemes to generate a look-ahead search tree from the current state. It uses a reward function which is based on an admissible heuristic and a collision avoidance estimate to estimate the reward for each transition that happens in the simulation model. At the current state of the world, the MCRT simulation model (defined in section 3.2.1.1 of this Chapter) uses a greedy action sampling scheme to select an action (applicable at the current state) to generate the look-ahead search tree. A stochastic transition function is used to compute the next state of the current state with the sampled action. At the next state, the simulation model selects one action as a sample using a random

sampling technique (described in detail in section 3.3 of this Chapter). The look-ahead search tree is extended from the next state to a fixed depth using a random sampling technique. The leaf node of the look-ahead search is evaluated using the reciprocal of the shortest distance estimate. The evaluation of the leaf node plus the sum of the reward of all transitions in a look-ahead tree is called the backup value of the tree. If the backup value of the tree is higher than the current value of the action, sampled at the current state, then the value of the current action is updated. This process of building the look-ahead search tree and computing the backup value in one simulation is called a rollout. The intuition behind the design of MCRT is discussed in section 3.1. The design of a rollout in MCRT is described in details in section 3.2.2. The formal notation and definitions are given in section 3.2. The algorithmic details of MCRT are described in section 3.3. Section 3.4 describes the complexity analysis of the MCRT algorithm. Two variations of MCRT are described in section 3.5. The convergence of a state in these algorithms is discussed in section 3.6.

3.1 MCRT: Selective Action Sampling Technique

MCRT is a policy rollout planning technique that runs multiple simulations from the current state and estimates the values of the actions applicable at the current state. MCRT generates the look-ahead search using a selective action sampling scheme. This means only one action is sampled at each state seen during the look-ahead search in a rollout. Selective action sampling greatly reduces the search time per simulation (or rollout) in a domain world of large state space when compared to non-selective action sampling approaches e.g. Sparse Sampling [35] (discussed in section 2.2.3 of

Chapter 2) where each action is sampled many times (at each state) in a simulation. Time per search is an important factor for a planner to solve planning problems in a RTS game as the environment demands a response within a constant (and very short) time interval.

There are three different ways to perform selective action sampling at a state. The first way is to randomly select an action (from the unexplored actions) applicable at a state. This is also known as exploration of new actions. This scheme is good for an incomplete information domain world because it can eventually find the best solution if it exists. However, the problem with this scheme is that it wastes computational time by exploring those actions which are not useful with respect to the solution of the current planning problem. The second scheme of performing selective action sampling (in a simulation) is a greedy action sampling scheme, i.e. always selecting the best action. This scheme, known as exploitation of the best actions, can lead to local maxima (like in hill-climbing) in an incomplete information domain world. The third scheme uses either exploration of new actions or exploitation of the best actions at a state in a simulation. The choice is made according to a given criterion. This criterion is used to find a balance between the exploration of new actions and the exploitation of the best action at a state. For example, in UCB the criterion is based on the number of times an action is sampled at a state (as shown in equation 2.5) and in UCT, the criterion is based on the number of times an action has been sampled at a state and also on a domain-specific constant called C_p (as shown in equation 2.9). These criteria are applicable in domains where the action values are in the range of [0, 1] (and also in the domain world where the action values can be transformed into this range (e.g.by using normalisation)). However, in domains like RTS games, action

values can be beyond this range and the maximum value of an action is not known (so the action values can not be transformed into the range [0, 1]).

The main contribution of MCRT is in the presentation of a new criterion to balance the trade-off between exploration of the new actions and the exploitation of the best actions. This criterion depends on the convergence of the action value. The convergence of an action value is defined with respect to a number (say n_c). If the maximum value of the best action at the current state does not change for n_c times, MCRT assumes that the action value is converged with respect to n_c and then this action is no longer selected as a sample at the current state in future simulations. A state is considered as converged if all of the actions (applicable at this state) are converged relative to n_c. MCRT does not perform rollouts for the converged state. The convergence of the action value and the state are discussed in more detail in section 3.6 of this Chapter. MCRT always applies the exploration of new actions at the states seen at the deeper levels of the look-ahead search tree.

3.1.1 Motivation for the MCRT Design

The main motivations for designing MCRT as a policy rollout algorithm are as follows.

1. A policy rollout algorithm keeps track of the action values (computed in the previous simulations), the number of times an action is sampled and the number of times a state is seen in previous simulations. These values can be used to tune the trade-off between the exploration of the new actions and exploitation of the best actions in a selective sampling scheme.

2. The use of selective action sampling is suitable for planning in real-time [39].

72

3. The policy rollout algorithm can be used to compute a good policy if a base policy (e.g. a heuristic function) is available in a domain world.

4. The look-ahead tree is implicitly generated, i.e. the algorithm does not explicitly store the look-ahead search tree in a simulation.

In MCRT, the depth of look-ahead search and the number of simulations are kept finite and fixed to guarantee action selection within a fixed time interval. This time interval is independent of the size of the state space. This characteristic makes MCRT select an action in real-time.

3.2   Notation

The formalisation of the planning problem is based on the assumption that the environment and the planning agent are two interacting parts of the domain world. The output of the planning agent is input to the environment and the output of the environment is the input to the planning agent. The output of the planning agent is an action which is an input to the environment. The action is executed by the environment and the effect of execution is an input to the planning agent (via its sensors). The problem formalisation also assumes that the environment is in the form of a grid. A cell can be empty or blocked by an obstacle. The obstacle can by dynamic or static. The size of the planning agent is exactly one cell. The planning agent has incomplete information about the environment as it does not know about the status of the cells (i.e. blocked or empty) which are beyond the visibility range of the planning agent. The planning agent also does not know when the status of a cell

(currently in the visibility range) can change due to a dynamic change. The latter sort of incompleteness of information makes the action effects non-deterministic.

The planning problem is represented as a tuple (S, A, T, R, s_o, G) where S is a finite set of states, A is a finite set of actions, T is a stochastic state transition function, $R(s, a)$ represents the reward value of taking action $a \in A(s)$ at state s, s_o is the initial state and G is the set of goal states. A state corresponds to a cell occupied by the planning agent. For any state $s \in S$, we define $A(s)$ to be the set of applicable actions at s where $A(s) \subset A$. The stochastic transition function T is a function of a state-action pair that randomly selects a next state for the input state-action pair. For example, for the current state s and action a, T selects a next state s_{next} randomly from a list of states (called $Next(s, a)$) that are possible to reach with a.

$$s_{next} = T(s, a) \tag{3.1}$$

The value of an action a at a state s is represented by $Q(s, a)$. $Q(s, a)$ for all $a \in A(s)$ is estimated by repeatedly running the simulations of MCRT from s. The details of these simulations are given in the algorithmic details of MCRT (section 3.3). In short, $Q(s, a)$ is estimated using the backup value (discussed in the start of this Chapter) of the look-ahead tree generated by the simulation model. The action $a \in A(s)$ that has the highest value when compared to other actions applicable at s is called the best action at s. The value of state s (represented by $V(s)$) is equal to the value of the best action (as shown in equation (3.2)). The set of immediate neighbours of s is represented by $Succ(s)$. The size of $Succ(s)$ is also equal to $|A(s)|$. There is a vector $Link(s, s') \, \forall s' \in Succ(s)$ associated with $Succ(s)$ such that for each $s' \in Succ(s)$, $Link(s, s')$ stores one and only one action $a \in A(s)$ that leads to s',

i.e. $a = Link(s, s')$. Figure 3.1 shows the algorithmic details of the population of $Succ(s)$ and $Link(s, s') \, \forall s' \in Succ(s)$ for the current state s. A state $s' \in Succ(s)$ is the best neighbouring state of s if $V(s')$ is greater than the value of any other state in $Succ(s)$.

Function $POPULATE - SUCC(s)$

1. $Link() := Empty;$

2. $Succ() := Empty;$

3.   Foreach $a \in A(s)$

4. $s' := T(s, a);$

5. $Succ.Add(s');$

6. $Link(s, s') := a;$

7.   End Foreach

8.   RETURN $[Succ, Link]$

End $POPULATE - SUCC$

Figure 3.1: Algorithmic details of finding the set of successor states.

$$V(s) = max_a Q(s, a) \tag{3.2}$$

A greedy policy π always selects the best action at a state s as a plan. In the case of a tie (i.e. more than one action has the same highest value at s), it randomly selects one action from the list of actions having the same highest value.

$$\pi(s) = \arg\max_a (Q(s, a)) \tag{3.3}$$

3.2.1   Definitions

The following definitions are used in the design of the MCRT technique.

3.2.1.1 Simulation Model

MCRT is applicable in a domain if a simulation model is available for this domain. This model can simulate the effects of an action, using the stochastic state transition function, and assign a reward for the state-action pair. The simulator has access to the sensor information that is the coordinates of the current state s and its successors ($\{Succ(s)\} \subset S$), the list of applicable actions ($A(s)$) at s, the list of the blocked states currently visible to the planning agent or those that the agent has seen before, the location of the goal states and the size of S. The simulation model also has information about the list ($Next(s, a) \subset S$) of the possible next states of a state s if an action $a \in A(s)$ is taken at s.

The simulation model can generate a look-ahead search tree of arbitrary depth d using the Monte-Carlo sampling techniques. We use a combination of two sampling approaches to design a rollout (section 3.2.2) in the simulation model of MCRT. The simulation model also uses the free space assumption, i.e., the states that are not visible or that have not been seen yet, are assumed to be empty spaces.

3.2.1.2 State Transition Probabilities

To estimate the next state $s' \in Next(s, a)$ of s by action a, the simulator uses a probability distribution P to prioritise the selection of the most likely transition. $P : S \times A \times S \mapsto [0, 1]$ is a probability density function that gives the probability of reaching a next state $s' \in Next(s, a)$ if an action $a \in A(s)$ is applied at state s. For example, if s_1, s_2 and s_3 are the possible next states of the current state s_c with action $a \in A(s_c)$ then $Next(s_c, a) = \{s_1, s_2, s_3\}$. Suppose the transition probabilities

are $P(s_c, a, s_1) = 0.2$, $P(s_c, a, s_2) = 0.6$, $P(s_c, a, s_3) = 0.2$. This means it is most likely that the planning agent would reach s_2 if a is applied at s_c. In MCRT, the simulator randomly selects a next state of s_c with high chances that it would be s_2. In MCRT rollout policy algorithm, the stochastic transition function T uses the transition probabilities P to compute the next state of a state-action pair.

3.2.1.3 Online State Transition Probabilities

In many domains e.g. RTS games, it is not possible to compute the state transition probability distribution offline prior to the start of the game as the game world is not known completely before the start of the game. We use a simple approach to compute and update/refine the state transition probability online. This approach assumes that all the transitions have the same probability initially. In other words, we can say that $P(s, a, s') = \epsilon$ for all $s' \in Next(s, a)$ where $\epsilon = \frac{1}{|Next(s,a)|}$ at the start of the planning process. The approach uses feedback from the environment to update the initial probabilities during planning and plan execution. It uses equation (3.4) to compute the probability distribution online. A transition probability $p(s_i, a, s_j) \in P$ represents the probability of moving an agent to s_j when $a \in A(s_i)$ is applied at s_i. At s_i, MCRT selects an action $a \in A(s_i)$ using equation (3.3). When the action is executed, the planning agent moves to s_j.

$$p(s_i, a, s_j) = N(s_i, a, s_j)/N(s_i, a) \tag{3.4}$$

where $N(s_i, a, s_j)$ is the total number of times action a is selected at s_i to move to s_j and $N(s_i, a)$ is the total number of times a is executed at s_i since the start of path planning for a given problem. The transition probabilities are updated based

on the actual transitions that have occurred in the system. The updated probability distribution is used in the simulation model. The exception to this is that a move to an *occupied* state has the probability 0. If any state $s' \in Next(s, a)$ is occupied by a static obstacle at any instance due to a dynamic change in the system, our approach updates P by setting $P(s, a, s') = 0$. For any state $s_j \in Next(s_i, a)$, $P(s_i, a, s_j) = 1$ if $s_j \in G$.

This approach is based on the idea of online estimation of the state transition probabilities in the generic indirect method [5] of solving a MDP with incomplete information.

3.2.2 Design of a Rollout

In a rollout of the MCRT simulation, MCRT generates the look-ahead search tree from the current state s by using two sampling techniques. To sample an action a at s (i.e. current state), MCRT finds a list of successor states of the current state i.e. $Succ(s)$. If each state $s' \in Succ(s)$ has been sampled before, then it selects the best neighbouring state as a sample otherwise it selects a neighbouring state randomly from the list of those states which have not been sampled in the previous simulations. Suppose $s_n \in Succ(s)$ is the sampled state then $a \in A(s)$ is selected as a sample where $a = Link(s, s_n)$. Then it finds the reward for the state-action pair (s, a) i.e. $R(s, a)$. At s_n, MCRT selects an action a_r randomly, computes the rewards for the state-action pair (s_n, a_r) and then finds the next state $s'_n = T(s_n, a_r)$. MCRT extends the look-ahead tree to depth d using random action sampling. Details of the random action sampling scheme are given in section 3.3. At the leaf node (s_d), MCRT estimates the shortest distance between s_d and the goal state g using an admissible

heuristic (h). The reciprocal of this heuristic value is used as an evaluation of the leaf node. Figure 3.2 shows an example rollout of MCRT. The backup value (R_n) is calculated by adding the reward values of all state-action pairs seen in the look-ahead search tree plus the evaluation of the leaf node. The value of the action sampled at the root node $Q(s, a)$ is set equal to the backup value R_n if an action is sampled for the first time. If a has been sampled in the previous simulations, then $Q(s, a)$ is updated if R_n is greater than the current value of $Q(s, a)$. In other words, $Q(s, a)$ increases monotonically.
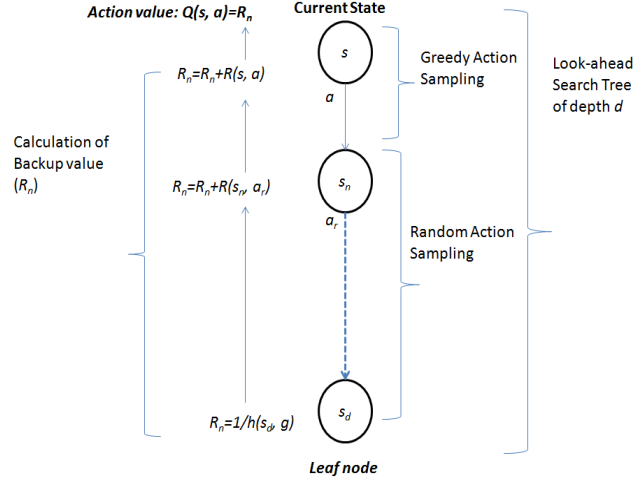
Figure 3.2: An Example of MCRT rollout with a look-ahead tree of depth d.

3.3   MCRT: Algorithmic Details

The MCRT technique takes the idea of the generic Monte-Carlo algorithm given in [39] and integrates it with the RRT random sampling technique [45] for applications in path planning. Based on the problem formulation, the current state (i.e. s_c),

and the current goal state (g), the MCRT function (Figure 3.3) returns the action it evaluates to be the best. It has access to system parameters that are calculated to take into account the real-time characteristics of the application to which MCRT is applied: the look-ahead depth, the time elapsed and its allowed cycle time. MCRT starts (line 1, Figure 3.3) by initialising the set of successor states and the link vector ($Link$). The repeat loop iterates as long as the real-time constraints allow. The $ChooseNeighbour$ function (line 3) selects which $s_n \in Succ(s_c)$ will be used for expansion: the choice is initially random from the set of unexpanded neighbouring states of state s_c, but once all neighbouring states have been seen it selects the neighbouring state with the highest state value currently recorded in $Succ(s_c)$. Then it finds the sample action a by using the link vector (line 4) i.e. $a = Link(s_c, s_n)$. After each call to the $RewardSim$ function is made (line 5), $Q(s, a)$ is updated with a new backup value if the backup value is greater than $Q(s, a)$ (line 6). After the simulations have finished, line 8 determines the best action at s_c and returns it for execution. The function $RewardSim$ (Figure 3.4) is adapted from [52]. This function estimates the reward of moving to s_n. $RewardSim$ expands a look-ahead tree from the neighbouring state s_n of the current state s_c, by generating random samples of states, and accumulating rewards as it searches, as explained below. The main recursive loop of $RewardSim$ starts in line 2 where a state position s_{rand} is chosen at random from any position on the map excluding (i) the agent's position (ii) the position of any obstacle that the agent can see. $RandomSample()$ (line 2) can also select the goal state as a sample. An action is selected to progress towards s_{rand} using a function called $SelectAction$. This finds the neighbour of s_n which is nearest to s_{rand} using the admissible heuristic, and selects an action which is aimed towards the

Function $MCRT(s_c, g)$

Read access $depth, timelimit$;

1. $POPULATE - SUCC(s_c)$; // find the successor states of s_c and populate Link vector

2.   REPEAT

3. $s_n := ChooseNeighbour(s_c)$;

4. $a := Link(s_c, s_n)$;

5. $r_n := R(s_c, a) + RewardSim(s_n, g, depth, 1)$;

6.   Update $Q(s, a)$ if $Q(s, a) < r_n$;

7.   UNTIL $(ElapsedTime() > timelimit)$;

8.   RETURN the best action a at s_c

End $MCRT$

Figure 3.3: High Level Design of MCRT.

Function $RewardSim(s_n, g, depth, d)$

Read access MDP;

1.   IF $d \neq depth$ THEN

2. $s_{rand} := RandomSample()$;

3. $a := SelectAction(s_n, s_{rand})$;

4. $[s_{next}, rw] := SimulateAction(s_n, a, g)$;

5.   RETURN $rw + RewardSim(s_{next}, g, depth, d + 1)$

6.   ELSE RETURN $1/dist(s_n, g)$

End $RewardSim$

Figure 3.4: RewardSim: MCRT Look-ahead Search

neighbour of s_n. A state s_{next} is generated that might be produced by the execution of this action in the $SimulateAction$ explained below, and the estimated reward of

that state is returned. The recursive call sums a series of rewards for each of the

advancing states, with a base case calculating the reward statically as the reciprocal

of the shortest distance from s to the target. The function $SimulateAction$ returns a

Function $SimulateAction(s_n, a, g)$

Read access MDP, W_d;

1. $s_{next} := T(s_n, a)$;

2. $rw := \|\{t : p(s_n, a, t)) > 0\}\| / (W_d * dist(s_{next}, g))$;

3. $return[s_{next}, rw]$

End $SimulateAction$

Figure 3.5: Simulate Action.

randomly selected state s_{next} that the agent may occupy after execution of action a,

and the estimate of the reward in moving to the new state s_{next}. The simulator uses

the probability distribution P to estimate the outcome of an action. A state s_{next} is

a possible new state after execution of action a at s_n if $p(s_n, a, s_{next}) > 0$. T (line 1 in

Figure 3.5) selects the next state s_{next} randomly from $Next(s_n, a)$, where the random

choice takes into account the probability that the state is reached by the execution of

a. Thus, the higher the chance a state would be reached, the more likely it is to be

chosen. In line 2 the reward for that state is calculated; the idea is that the larger the

list of possible new states is for an action, then the higher the reward. This is based on

the intuition that a larger list indicates there are likely to be fewer obstacles present

in that action's direction of travel (and thus reduces the chance of future collisions).

The size of the list is divided by $dist(s_{next}, g)$, the shortest distance estimate to the

goal g from the new state s_{next}, as the further away, the less the reward. Finally the reward is given a scaling factor W_d which normalises the relationship between the collision-free path and distance to goal: for a particular application of MCRT, this would be tuned to balance the importance of directing towards collision free paths while minimising the shortest distance to goal states.
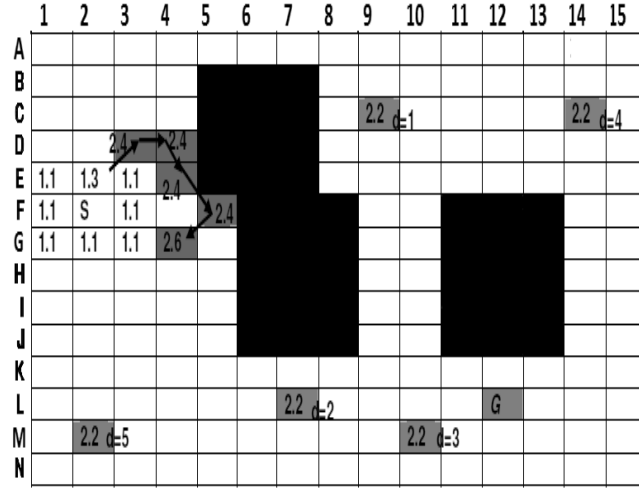
Figure 3.6: A MCRT Example: S is the current state, G is the goal state and d is the look-ahead depth. The cells in Black are static obstacles.

Figure 3.6 shows a grid example of MCRT search with look-ahead of depth 5. The search starts at S and determines the immediate neighbours of S (line 1, Figure 3.3). These neighbours are labeled as 1.1 in Figure (3.6) (in Figure 3.6, the label X.Y means that the state is produced by Figure X in line Y). If S is seen for the first time, then a neighbour is selected randomly (line 3, Figure (3.3)). Suppose the randomly chosen neighbour is E2. Then MCRT runs MC simulations ($RewardSim$) to estimate the reward for the transition from S to E2. $RewardSim$ expands the

look-ahead search using the randomly exploring sampling approach. A state, say C9, is selected randomly at look-ahead depth $d = 1$ from the state space (according to line 2 of Figure (3.4)). An action a is selected to expand E2 towards C9. The action a is simulated (line 4, Figure (3.4)) and an outcome of a is estimated using the probability distribution. In the example, D3 is assumed as the estimated next state of E2 when action a is applied. The simulation also estimates and stores a reward for the state-action pair i.e. (E2, a). The depth of the look-ahead search is increased and $RewardSim$ is run from the next state i.e. D3. An action a at D3 is selected according to the second random sample, say L7, and simulated to estimate the reward and the next state of D3 with action a, and this reward is added to the previously stored reward. This process continues until look-ahead search reaches a depth of five. The next state at depth five is a leaf node of the look-ahead search. In Figure (3.6), G4 is the leaf node of the look-ahead search, and is evaluated using line 6 of Figure (3.4). The evaluated value is added to the accumulated reward and used as the final value of the reward for the transition from S to E2. The simulations are continued for the maximum allowed time or when the convergence condition is satisfied.

3.4 Complexity Analysis

The complexity of MCRT is discussed per simulation. In each simulation, MCRT look-ahead search is expanded only by a fixed depth d. To explain the time complexity of one simulation, we discuss the time complexity of each step of generating the look-ahead search tree in MCRT. The MCRT look-ahead search starts by finding a set of neighbouring states ($Succ(s)$) of the current state s. This step has the time and

space complexity of $O(|A(s)|)$. The selection of a neighbouring state $s_n \in Succ(s)$ as a sample is $O(1)$ and $Link(s, s_n)$ is $O(1)$. $RewardSim$ expands the look-ahead search from the neighbouring state s_n by using a random state $s_r \in S$. The computing of s_r has the time complexity of $O(1)$. To find an action $a \in A(s_n)$, MCRT evaluates each action $a \in A(s_n)$ that has the highest reward $R(s_n, a)$ for moving towards s_r as a goal. The worst-case complexity of $SelectAction$ (line 3, Figure 3.4) is $O(|A|)$. The time complexity of the function $SimulateAction$ (Figure 3.4) is $O(1)$. $RewardSim$ computes $d - 1$ samples in a simulation, so the time complexity of $RewardSim$ is $O((d-1) \times |A|)$. The worst-case time complexity of MCRT per simulation for a root state s is $O(d \times |A|)$. The worst-case time complexity of MCRT for K simulations is $O(Kd|A|)$. The worst-case time complexity of MCRT is independent of the size of the state space.

MCRT stores all the neighbours of the current state and also random states in a simulation. The random states are saved to avoid sampling the same state more than once. The worst-case space complexity of MCRT is $O(K(|A| + d))$ for K simulations.

3.5 MCRT Variations

MCRT design is very flexible in that it allows the use of different sampling schemes for exploration and exploitation of the promising actions. In this section, we describe two variations of MCRT: Random Action Sampling (MCRT-RAS) and Corridor based Action Sampling (MCRT-CAS). In Random Walk, MCRT generates the look-ahead search using the random selection of an action $a \in A(s)$ at each state s seen in $RewardSim$ (Figure 3.4) without using the random samples of the states. In the

other approach, i.e. MCRT-CAS, it selects a greedy action at the current state as a sample and the successor states use a combination of exploitation and exploration based action sampling. MCRT-CAS explores the new actions within a small list of actions.

3.5.1 MCRT with Random Action Sampling

MCRT with Random Action Sampling (MCRT-RAS) is a modified form of MCRT where the actions are sampled randomly at the successor states of the current state during the look-ahead search. This sampling scheme is applied in the *RewardSim* function of MCRT. In MCRT-RAS, at the current state the sampling approach is the same as in the original MCRT. In other words, MCRT-RAS uses a different exploration based sampling from the RRT sampling of the original MCRT. The random action selection is faster than the random exploration of MCRT. It speeds up the generation of the look-ahead search of depth d in a simulation because MCRT-RAS does plan an action to a random state.

3.5.1.1 Algorithmic Details

MCRT-RAS is similar to MCRT except for the random sampling in the *RewardSim* function. The modified *RewardSim* is shown in Figure 3.7. At any state s_n, if it is not a leaf node (line 1), it is expanded by *RewardSim*. In this case, MCRT-RAS randomly selects an action a from all applicable actions ($A(s_n)$) at s_n (line 2, Figure 3.7). This action is simulated by *SimulateAction* (line 4, Figure 3.7) and a next state of s_n is computed by this function with the sampled action a. This function also estimates an immediate reward for this state transition. If s_n is a leaf node, it is

evaluated by a heuristic function (line 5, Figure 3.7) rather than expanding it.

Function $RewardSim(s_n, g, depth, d)$

Read access MDP;

1.   IF $d \neq depth$ THEN

2. $a := RandomSelection(A(s_n))$;

3. $[s_{next}, rw] := SimulateAction(s_n, a, g)$;

4.   RETURN $rw + RewardSim(s_{next}, g, depth, d+1)$

5.   ELSE RETURN $1/dist(s_n, g)$

End $RewardSim$

Figure 3.7: RewardSim: MCRT-RAS

3.5.1.2 Complexity Analysis

MCRT-RAS has better space complexity than the original MCRT because MCRT-RAS does not store the random state samples in a simulation. The worst-case space complexity of MCRT-RAS is $O(|A|)$ per simulation. The time complexity of MCRT-RAS is discussed in a stepwise manner according to Figure 3.7. The random selection of actions $RandomSelection(s)$ (line 2) at state s has the time complexity of $O(|A|)$. The simulation of an action a at s takes $O(1)$. The evaluation of the leaf node of the look-ahead tree in a simulation is of order $O(1)$. As $RewardSim$ expands the look-ahead tree by $d - 1$ so it has the time complexity of $O(d - 1)$. The worst-case time complexity of MCRT-RAS per simulation is $O(|A| \times d)$.

3.5.2 MCRT-Corridor Based Action Sampling (MCRT-CAS)

MCRT-CAS expands the look-ahead search from the current state s to a fixed depth D in a simulation. At s, it chooses the best action $a_s \in A(s)$ as a sample. It explores the actions at the successor state $s' = T(s, a_s)$ of s seen in the look-ahead search. At s', it selects an action $a' \in A(s')$ from a small list of actions $A_c(s') \subset A(s')$ such that $a' \in A_c(s')$. The small list of actions is called a corridor. $A_c(s')$ is constructed using the best action a at s'. This sampling continues until the look-ahead search reaches depth d. For each state-action pair seen during the look-ahead search, MCRT-CAS keeps computing the reward values using the R function. The state seen at depth d is evaluated using an admissible heuristic and this value is added to the sum of the rewards of all state-action pairs seen in the look-ahead search in a simulation. This accumulated value is used to update the expected long term reward of the action sampled at the current state, i.e. $Q(s, a_s)$. If the returned value is bigger than the current $Q(s, a_s)$ then the estimated action value is modified otherwise it remains unchanged. If $Q(s, a_s)$ remains unchanged for $n_c > 0$ consecutive simulations, then MCRT-CAS selects the best action at s by excluding a_s from the action list. If s has converged or if the time to run the simulations expires, it selects the best action at s as a plan and returns a for execution.

3.5.2.1 Objective

In Monte-Carlo simulations, the exploration of new actions is an important and essential part of the search. It is important because the action values at the current state of the planning agent are estimated using local information. However, the ex-

ploration of new actions that are not useful to solve the current planning problem is computationally very expensive for the planning search algorithm. It is intuitive to limit the exploration of the search space - during the Monte-Carlo simulations - within the vicinity (or corridor) of the effect(s) of the current best action of a state. The exploration scheme also gives importance to the actions that are less explored by increasing their chance of selection. This ensures the exploration of all possible actions in a corridor. The chance of selection is computed using $1/n(s, a)$ where $n(s, a)$ is the number of times an action $a \in A(s)$ is sampled at state s. The corridors are kept overlapping, i.e. two corridors (each one by a different action) can have one or more common actions, so the exploration of new actions can move from one corridor to another one in two consecutive simulations. At the current state s, the exploration of a new action is performed if the estimated value of the best action $a_b \in A(s)$ does not change for some iterations.

3.5.2.2 Algorithmic Details

MCRT-CAS makes two main changes to the original MCRT [52]. First it uses the estimated action values to draw a sample at the current state of the planning agent. Second, it uses a corridor-based exploration scheme to draw action samples at the successor states of the current state. To explore the new actions, MCRT-CAS uses a small set of actions that are relevant to the best action in the current state. MCRT-CAS performs exploration of new actions within that corridor. The construction of the corridor for an action is done automatically. This can be done by using the angle between the directional lines of two actions if the actions are directional. A corridor for an action is built before the start of planning and is stored in memory for online

use. The construction of a corridor for an action is a trivial process. We use the angle between the directions of two actions to build a corridor. For example, a corridor of an action "MOVE TO NORTH" is a set {"MOVE TO NORTH WEST", "MOVE TO NORTH", "MOVE TO NORTH EAST"} where each member of the set has an angle of 45° or less with the action "MOVE TO NORTH". The general overview of MCRT-CAS is given in Figure 3.8. At the current state s_c, if s_c has converged then MCRT-CAS returns the best action at s_c (line 2, Figure 3.8). The convergence of a state s_c is discussed in section 3.6. If s_c has not converged yet, then MCRT-CAS runs several rollouts depending on the time limit to estimate the action values at s_c. In each rollout, MCRT-CAS chooses (line 6, Figure 3.8) the best action $a \in A(s_c)$ at s_c as a sample. If s_c is seen for the first time, or any of its actions have not been sampled yet, then $ChoooseAction$ randomly selects an action (from the unseen actions). The next state s_n of s_c along with the reward (r_n) of the state-action pair (s_c, a) is returned by the function $SimulateAction$ (line 7, Figure 3.8). The details of $SimulateAction$ have been discussed in section 3.3. s_n is expanded for a length of $depth - 1$ using a combination of the corridor-based action sampling. The function $ChooseActionFromCorridor$ (line 9, Figure 3.8) chooses the best action a' at s_n and creates a corridor $A_c(a')$. The function then selects an action a randomly from the corridor of a'. The chance of selection of an action as a sample in the corridor depends on the number of times the action has been sampled in the previous searches. An action $a \in A_c(a')$ has more chance of selection as a sample than other members of $A_c(a')$ if a is the least explored. The function $SimulateAction$ (line 10) computes the immediate reward r_w and the next state s_{next} of the state action pair (s_n, a). r_w is added to r_n (line 11). s_{next} is expanded in the same way that s_n is done. This

Function $MCRT - CAS(s_c, g)$

Read access $depth, timelimit, n_c$;

1.   IF s_c is converged THEN

2. $a := ChooseBestAction(s_c)$;

3.   RETURN the action a;

4.   ELSE

5.   REPEAT

6. $a := ChooseAction(s_c)$;

7. $[s_n, r_n] := SimulateAction(s_c, a, g)$;

8.   FOR: i=1 to depth -1

9. $a := ChooseActionFromCorridor(s_n)$;

10. $[s_{next}, r_w] := SimulateAction(s_n, a, g)$;

11. $r_n := r_n + r_w$;

12. $s_n := s_{next}$;

13.   END FOR

14. $r_n := r_n + 1/dist(s_n, g)$;

15.   IF $Q(s_c, a) < r_n$ THEN

16. $Q(s_c, a) := r_n$;

17. $timelimit = timelimt - 1$;

18.   UNTIL $(timelimit > 0)$;

19. $a := ChooseBestAction(s_c)$;

20.   RETURN the action a

End $MCRT - CAS$

Figure 3.8: High Level Design of MCRT-CAS.

continues for $depth - 1$ iterations. At depth $depth$ of the look-ahead search, the leaf node s_n is evaluated using the distance heuristic $dist$ and the inverse of the heuristic

is added to r_n (line 14). If the current estimate of the long term reward r_n of the state action (s_c, a) is greater than or equal to the previous value, i.e. $Q(s_c, a)$, then MCRT-CAS updates $Q(s_c, a)$ (line 16). MCRT-CAS keeps running the simulations - with s_c as the start node for each simulation - until the time is out. At the end of the simulations, MCRT-CAS selects the best action at s_c (line 19) and returns it for execution.

3.5.2.3 Complexity Analysis

MCRT-CAS selects an action from current state s_c using *ChooseAction* (lines 6, Figure 3.8). It takes $O(1)$ to select an action randomly at s_c if it is seen for the first time. If all actions at s_c have been sampled in previous searching efforts, then it takes $O(|A(s_c)|)$ to select the best action at s_c (line 1, Figure 3.8). The average time complexity of *ChooseAction* is $O(1)$ and the worst-case time complexity is $O(|A|)$. The worst-case time complexity of *ChooseActionFromCorridor* (line 9, Figure 3.8) is $O(C)$ where C is the size of the corridor. The worst-case space complexity of MCRT-CAS per simulation is $O(d + C|A|)$. It has a worst-case time complexity of $O(|A| + (d - 1)C)$. The MCRT-CAS technique has the worst-case time complexity of $O(K(|A| + (d - 1)C))$ if K is the number of rollouts. MCRT-CAS is better than MCRT and MCRT-RAS in terms of the worst-case time complexity.

3.6 Convergence

From Figure 3.8 (line 16), it is obvious that $Q(s, a)$ for any state $s \in S$ and action $a \in A(s)$ remains the same or increases with the increase in the number of rollouts.

At any simulation time $t > 1$, the $Q_t(s, a)$, $\forall s \in S$, $a \in A(s)$ is monotonically non-decreasing as shown in Equation 3.5.

$$0 < Q_{t-1}(s, a) \leq Q_t(s, a), \ \forall t > 1 \tag{3.5}$$

MCRT-CAS also preserves the monotonicity of the state value $V(s)$ for each state $s \in S$. The optimal state value $V^*(s)$ for state $s \in S$ is the supremum of the monotonic sequence $V(s)$ and the optimal action value $Q^*(s, a)$ is $sup(Q(s, a))$ for $a \in A(s)$. In other words, the optimal action value of a state action pair (s, a) is an upper bound on the monotonic sequence of $Q_t(s, a)$ for all $t > 1$ (Equation (3.5)).

Lemma 1. *A monotonic function (of real numbers) converges if it is bounded.* [23]

THEOREM 1. *MCRT-CAS eventually finds the optimal action value $Q(s, a)$ for a given state s and action $a \in A(s)$ if repeated for several iterations.*

Proof:

The following assumptions are used to prove the theorem.

1. The domain world is static.

2. The action value at any time t during the simulations can never be greater than the optimal value (i.e. optimal value is an upper bound on the action value).

The proof follows Lemma 1. MCRT-CAS generates a monotonic sequence $Q(s, a)$ of action values for the state action pair (s, a) by running several simulations from $s \in S$. It is given that the optimal value $Q^*(s, a)$ is an upper bound of the monotonic sequence $Q(s, a)$. Therefore, MCRT-CAS eventually converges to the optimal action

value for a given state action pair (s, a). $\square$

It is not clear how many iterations MCRT-CAS requires to converge the action value function $Q(s, a)$ to an optimal value for a given state s and action a. For practical reasons, we use an error bound based on the algorithmic parameter $n_c > 0$ to define the convergence of the action value function Q. If MCRT-CAS does not change the action value of a state-action pair (s, a) for n_c consecutive simulations, then it is assumed that $Q(s, a)$ has converged relative to parameter n_c. If all applicable actions $a \in A(s)$ at s are converged with respect to n_c then s is declared a converged state relative to n_c. For higher values of n_c, the error in the estimation of action values should reduce. Theorem 1 is also true for MCRT and MCRT-RAS.

The convergence of a state assumes that the domain world is static. In a dynamic world, the set of applicable actions can change anytime during problem solving. In the Monte-Carlo techniques (MCRT, MCRT-RAS and MCRT-CAS), the value of an action is set to zero if the action becomes inapplicable at a state. The main reason for decreasing the value of an action to zero (if it becomes inapplicable at any state) is that the value of this action must be re-computed (if the same action becomes applicable) in future. If any action of a converged state becomes inapplicable or an inapplicable action becomes applicable (during problem solving), then that state becomes a non-converged state.

3.7 Summary

In this chapter, we have described a Monte-Carlo planning technique called MCRT that uses a novel way of balancing the exploration of new actions and the exploitation

of promising actions in the simulations. The time per simulation in this approach is independent of the size of the state space in a planning problem. We presented two variations of MCRT, one reduces the space complexity of MCRT (i.e. MCRT-RAS) and the other speeds up exploration by using a small region around the best action (at a state) according to the current estimates of the action values. The space and time complexity of all three algorithms were discussed per simulation. The action values at the current state of the domain world are learned (or estimated) by running several simulations (or rollouts) within an allowable time. Each rollout starts from the current state and ends when the look-ahead search reaches a finite depth. Each rollout estimates a value for an action of the current-state. The actions at the current state are always selected using the greedy approach (i.e. the best action). The estimated action values monotonically increase with the increase in the simulation time unless the state is converged. If a current state has converged, then MCRT or its variations do not expand it, instead, these algorithms select the best action and return it for execution.

MCRT and its variations are suitable for real-time planning in general and for real-time path planning in particular. These algorithms are suitable for real-time planners that interleave planning and execution and run under tight time constraints.

Chapter 4

MOCART Planners

In this chapter, two kinds of Monte-Carlo planner are presented that exploit the MCRT technique and its variations (MCRT-RAS and MCRT-CAS) to solve path planning problems under real-time constraints and partial visibility. The first kind of planner is designed to solve single journey path planning problems. There are three planners in this category. These planners interleave planning and plan execution to solve a planning problem under tight time constraints. The algorithmic details and the properties (i.e. completeness and correctness) of these three planners are described in section 4.1. The second kind of planner is a real-time Monte-Carlo path planner that solves multiple journey path planning problems. A multiple journey path planning problem has more than one goal. This planner, along with the motivation for defining a planning problem with multiple goals, is described in section 4.2. The planner in the second category is called a Multi-Goal MOCART (MG-MOCART) planner.

4.1 Monte-Carlo Real-Time Path Planners

In this section, we present three planners that are based on the three Monte-Carlo techniques (i.e. MCRT, MCRT-RAS and MCRT-CAS) presented in Chapter 3. Each of these planners chooses an action (using one of these three techniques) at the current state of the domain world as a partial path plan, executes the selected action and moves the planning agent to a new position. At the new position the planner repeats the same process, i.e. interleaving planning and plan execution if the new position is not a goal location. This sequence of planning and plan execution continues until the planning agent reaches the goal location. The first planner uses the MCRT technique to select an action at the current state of the domain world. This planner is called a MOCART planner. The second planner is called a MOCART-RAS planner because it uses the MCRT-RAS technique for the action selection at the current state. The third planner is called a MOCART-CAS planner as it uses the MCRT-CAS technique to plan an action at the current state of the domain world. The algorithmic details of MOCART planner are given in Figure 4.1.

The planner reads a planning problem with the initial state s_o of the planning agent and the goal state g it intends to move to. The planner initialises the MOCART parameters (line 1, Figure 4.1). These are parameters *timelimit*, n_c and look-ahead depth *depth*. The planner calls the MCRT technique at the initial state s_o to select an action $a \in A(s_o)$ to move towards g (line 3). The action selected by MCRT is executed at s_o and the agent moves to a new state s' (line 4). The state transition probabilities are updated for the transition (line 7). If s is a goal state then the planner stops otherwise it keeps planning and executing until the planning agent

Procedure *MOCARTplanner*

Read(s_o, g);

1.   initialise parameters and state $s := s_o$;

2.   REPEAT

3. $a := MCRT(s, g)$;

4. $s' := Execute(a, s)$;

5. $n_e(s, a) = n_e(s, a)+1$;

6. $n_e(s, a, s') = n_e(s, a, s')+1$;

7. $P(s, a, s') = \frac{n_e(s,a,s')}{n_e(s,a)}$;

8. $s := s'$;

9.   UNTIL $s.pos = g$;

End *MOCART − Planner*

Figure 4.1: MOCART planner

finds the goal state. The MOCART planner keeps track of $n(s, a)$, i.e the number of times action a is executed at state s (line 5). The planner also stores the number of times the domain world changes from state s to s' when action a is taken at s (line 6). These statistics are maintained for two reasons. First, these numbers are used to update the state transition probabilities. The probability distribution is updated using Equation (3.4, chapter 3). The second reason for maintaining these statistics is to impose a limit on the number of times an action a is executed at state s. If $n(s, a)$ is greater than a user-specified number $n > 0$, then a is not executed at s even if it is the best action at s. This restriction is used to ensure completeness of the MOCART planner. The MOCART planner will select the next best action at s if the best action a at state s has $n(s, a) > n$. This limit is also used to guarantee the completeness of

the planner. The completeness of the MOCART planner is discussed in section 4.1.1. The MOCART-RAS and MOCART-CAS planners are the same as the MOCART planner except for the use of the Monte-Carlo technique in line 3 of Figure 4.1. In the MOCART-RAS planner, line 3 uses the MCRT-RAS technique and the planner is a MOCART-CAS planner if the MCRT-CAS technique is used in line 3 of Figure 4.1.

4.1.1 Completeness

In this section, the completeness of the MOCART Planner is described in detail. The same description is also true for the MOCART-RAS and MOCART-CAS planners. In every state, the MOCART planner decides which action will be executed exploiting the local information around the state. This section raises a concern whether using limited local information leads towards solving a global task (reach the goal state). In a finite state and action space, there exists only one reason why the planner might not reach the goal state which is that the agent is moving in a cycle. The cycle can be broken by limiting the number of executions of an action in a state. In a static and partially observable environment, this ensures completeness (it is discussed in detail in **Theorem 2**). In dynamic worlds, this property does not ensure completeness. However, this is a common property of all dynamic search algorithms [11]. Following the tradition of ensuring completeness of a dynamic search algorithm in a static world (e.g. Real-Time D* [11] ensures completeness in the static world only), we also highlight completeness of the MOCART planning by assuming that the domain world is static and partially observable.

Theorem 2: *In a finite state space S with a finite set of actions A, in which a goal state is reachable from every state of the domain world, the MOCART Planner will eventually reach the goal state.*

Proof:

For the sake of simplicity, we make the following assumptions about the domain world.

1. The domain world is static.

2. The environment is partially observable.

3. For every state $s \in S$ and a successor state $s' = T(s, a)$, there is always an action $a' \in A(s')$ such that $s = T(s', a')$.

To prove that the MOCART Planner will always reach the goal state g from any initial state $s_o \in S$, we suppose that a path exists between s_o and g and the MOCART planner never reaches g. Since there is no one-way edge leading to a dead-end (according to assumption 3), the only possible reason for never reaching a goal state is if the planning agent keeps moving in a cycle forever. The cycle is a subset of the state space and has a finite number of states, as the size of the state space is finite. At each state s, there should be at least one action a such that execution of this action at state s keeps the planning agent moving in the cycle. We assume the MOCART planner always selects this action $a \in A(s)$ at s to execute. However, it is given that the MOCART planner can select an action a at s for execution for a fixed number of times, i.e $n > 0$. After n executions of a at s, the planner selects another action $a' \in A(s)$ to move the planning agent. If a' also keeps the planning agent moving in a cycle, then a' is selected for a maximum of n times to execute at s. This process

continues until the MOCART planner eventually executes each action $a \in A(s)$ at least once. Since the goal is reachable from every state, then s must have at least one action $a_{es} \in A(s)$ that can make the planning agent get out of the cycle. As each action is executed at least once, then it is guaranteed that a_{es} is eventually executed at s. After the execution of a_{es} at s, the planning agent moves out of the cycle and moves towards the goal state. So, the MOCART planner can break the cycle.

There is a possibility that the planning agent can re-enter the region of an already broken cycle. In this case, suppose $s \in S$ is a part of the cycle such that all actions applicable at s have been executed previously. In this scenario, the MOCART planner selects an action that has been executed for the minimum number of times as compared to other applicable actions at s. Ties are broken randomly. This action selection mechanism ensures that each action applicable at s is executed in this scenario. As it is given that s has at least one action that leads it out of the cycle. Therefore, the MOCART planner always break the cycles. This contradicts the assumption that the planner never reaches the goal state, therefore, the MOCART planner will eventually reach the goal state. □

4.1.2 Correctness

We use a notion of correctness that has been used for a real-time planner (LRTA) in [43]. According to this notion of correctness, an action selected by the planner (at the current state of the domain world) is correct if the action is valid (i.e. executable) and execution of this action moves a planning agent towards the goal location.

The MOCART planner applies local search with random sampling to estimate the

action values. The selection of an action (for execution) is based on these estimates. In this section, we raise the concern whether the planner selects the correct action. Execution of a correct action, at any state, moves the planning agent towards the goal state (i.e. reduces the distance between the current location of the planning agent and the goal location). The question of reaching the goal state has been discussed in the previous section. This section highlights the elements of the MOCART planner that ensure correctness relative to the neighbourhood of the current state. The correctness of the MOCART planner also assumes that there are no cycles. The same assumption has been used in a previous work [43] in defining the correctness of Learning Real-Time A*. In the case of a cycle, the local search planners like LRTA, RTD and the MOCART planner choose actions to escape the cycle (which is contrary to the definition of correctness) to ensure completeness.

Action selection (for execution) is based on the values of the actions estimated by gathering information from the neighbourhood of the current state. It chooses the best action according to the current estimates of actions applicable at the current state. The best action is an optimal action relative to the neighbourhood of the current state. The action values are estimated using a reward function that uses an admissible heuristic function to estimate the distance of the effect of the best action from the goal state. The long term reward of an action a at state s is estimated by generating a look-ahead search tree (of fixed depth d) from s to a fixed horizon d. This look-ahead search tree can be considered as a partial path of length d from the current position s towards the goal position g. The total reward of this trajectory is called the *path reward* and is equivalent to $Q(s, a)$. A partial path that has the highest path reward is the shortest collision avoidance path to the goal state. To define the

correctness of the MOCART planner, we assume that actions are directional and the effect of an action is always along the direction of the action (for example, if an action "Move to South" is applicable at the current state, then all possible next states of the current state with this action are in the south direction only). $h(s, g)$ represents the estimate of the shortest path between state s and goal state g using the free space assumption. $c_e(s, a)$ represents the collision avoidance estimate for the state-action pair (s, a) (i.e. $c_e(s, a) = |Next(s, a)|$). The set of successor states of a state s is represented by $Succ(s)$. We denote the path reward (of a partial path of length d) by $\rho(d)$. In this section, the correctness of the MOCART and MOCART-CAS planners is described. Details of the correctness of the MOCART planner are also true for the MOCART-RAS planner because both planners have the same sampling approach at the current state of the domain world in a simulation.

4.1.2.1 Correctness of the MOCART planner

Theorem 3: *The MOCART planner always selects an action a at the current state $s \in S$ such that the effect $(s' = T(s, a))$ of this action reduces the distance between the planning agent and the goal state g with respect to the current neighbourhood of s and also avoids collision with static obstacles.*

Proof:

The theorem proof uses the following assumptions for the sake of simplicity.

1. s is not a converged state.

2. There are are no cycles in S.

3. The domain world is static.

4. The environment is partially observable.

5. The states which are not seen by the planner are assumed to be empty (i.e. free space assumption).

Action selection (by the MOCART planner) depends on the estimated value of the actions applicable at the current state of the domain world and these estimates are computed using the MCRT technique with a fixed number of Monte-Carlo rollouts and a look-ahead search of depth d. Therefore, the proof of Theorem 3 depends on the number of rollouts and the look-ahead depth d. For this purpose we use Proposition 1 to prove the theorem by applying mathematical induction on this proposition.

Proposition 1: *If a is the best action at state s according to the current estimates of the action values computed (by the MCRT technique) in any t simulations, then $s' = T(s, a)$ is on the shortest path of length d to g relative to the neighbourhood of s. d is the depth of the look-ahead search in each simulation.*

Base Case ($t = 1$): If $t = 1$, the MCRT technique uses the neighbourhood ($Succ(s)$) of s to select an action at s for execution. It computes state values $V(s')$ for each neighbouring state $s' \in Succ(s)$. $V(s')$ is computed using $\frac{1}{h(s',g)}$.

The best neighbouring state s' is a state that has the highest state value when compared to other neighbouring states of s. The MCRT technique then selects an action a at s such that it can move the planning agent from s to the best state s', i.e $s' = T(s, a)$. The state value of s' has the lowest $h(s', g)$ when compared to other states in $Succ(s)$. Therefore, s' is on the shortest path to the goal state g with respect to other states in $Succ(s)$. $Proposition1$ is true for $t = 1$.

Induction Case ($t = k$ and $d \geq 1$): Suppose, Proposition 1 is also true for $t = k$. This means the MCRT technique finds a best neighbouring state s' at s with k rollouts from s and a is the action that can move the planning agent from s to the best

neighbouring state (s'). The partial path of length d has the reward value equal to $Q(s, a)$ (i.e. $\rho_k(d) = Q(s, a)$). At $k + 1$ rollout, the MCRT technique can have one of three possible cases. In the first case, at $t = k + 1$ rollout, the MCRT technique can find a new backup value that is greater than $Q(s, a)$. $Q(s, a)$ is updated. In the second case, $Q(s, a)$ remains unchanged because the backup value is smaller than $Q(s, a)$. In both cases, a remains the best action at s and s' the best neighbouring state. So Proposition 1 is true in the first and second case for $t = k+1$. The third case arises when $Q(s, a)$ has converged relative to a parameter n_c (known as the convergence parameter). If $Q(s, a)$ has converged relative to n_c at the kth simulation, then at $k + 1$ iteration, the MCRT technique selects another action $a_r \in A(s)$ such that $a_r \neq a'$ at s. The neighbouring state $s_r = T(s, a_r) \in Succ(s)$ is selected to expand the look-ahead search. If $Q(s, a_r) > Q(s, a')$ then this also implies that $\rho_{k+1}(d) > \rho_k(d)$. Therefore, the new best state s_r is on the shortest path relative to the current neighbourhood of s. This makes P true if a new partial path (of greater reward than the previous one) is discovered. If $Q(s, a_r) < Q(s, a')$ then Proposition 1 is also true as s' is on the shortest path of length d discovered so far. Hence, Proposition 1 is also true for $t = k + 1$ in all possible cases. $\square$

4.1.2.2 Correctness of the MOCART-CAS Planner

Theorem 4: *The MOCART-CAS planner selects an action a at the current state $s \in S$ such that the effect $(s' = T(s, a))$ of this action reduces the distance between the planning agent and the goal state g and also avoids collision with the static obstacles.*

Proof:

The theorem proof uses the following assumptions for the sake of simplicity.

1. s is not a converged state.

2. There are no cycles in S.

3. The domain world is static.

4. The environment is partially observable.

5. The states which are not seen by the planner are assumed to be empty (i.e. free space assumption).

Proposition 2 is used to prove the correctness of the MOCART-CAS planner.

Proposition 2: *If a is the best action at state s according to the current estimates of the action values computed in any t simulations , then $s' = T(s, a)$ is on the shortest collision avoidance path of length d to the goal state g relative to the neighbourhood of s. d is the depth of the look-ahead search.*

Base Case ($t = 1$): If $t = 1$, the MCRT-CAS technique computes the best action a at s using reward function $R(s, a)$ with the look-ahead search of depth 1, i.e. $d = 1$. The best action a has the maximum value of $R(s, a)$ at s (Equation 4.1).

$$a = \arg\max_a(R(s, a)). \tag{4.1}$$

$R(s, a)$ for action a depends on $h(s', g)$ and $c_e(s, a)$ where $s' = T(s, a)$ is the next state of s. $c_e(s, a)$ considers the collision effects (to avoid moving towards the dead-ends). $R(s, a)$ is maximum if $h(s', g)$ is minimum and $c_e(s, a)$ is maximum. It is given that the reward function does not use $c_e(s, a)$ in the case $c_e(s, a) > h(s', g)$. This is the case when s is very close to the goal state (g) and there is no obstacle between s and g. For example, if the next state of s with action a is at a distance of one from the goal state (i.e. $h(T(s, a), g) = 1$) and $c_e(s, a) = 2$. Suppose there is another action $a' \in A(s)$ at s such that it has $h(T(s, a'), g) = 2$ and $c_e(s, a) = 5$. $R(s, a')$ can be

106

higher than $R(s, a)$ even if a leads to the goal state. The correct value of $R(s, a)$ is computed by ignoring $c_e(s, a)$ in this case. In this case, the best action a at s with $t = 1$ has the minimum value of $h(T(s, a), g)$ as compared to other actions applicable at s. Therefore, Proposition 2 is true in this case.

Suppose another scenario where a is the best action at s and $c_e(s, a) < h(T(s, a), g)$ and there is an action $a' \in A(s)$ such that $h(T(s, a'), g)$ is smaller than $h(T(s, a), g)$ but $c_e(s, a) > c_e(s, a')$. In this case, $R(s, a)$ is greater than $R(s, a')$. In this case, $R(s, a)$ represents the reward value $(\rho(d))$ of the shortest collision avoidance path of length d from s towards the goal state g. Therefore, Proposition 2 is true at $t = 1$ and $d = 1$; this is also true for $t < |A(s)|$.

Induction Case ($t = k$ and $d \geq 1$): We assume the MCRT-CAS technique also selects a best action a' at s by estimating the action values with k simulations (or rollout) and look-ahead depth of d such that $s' = T(s, a')$ is on the shortest path of length d to g from s. Proposition 2 is true for $t = k$ where $k > |A(s)|$. The shortest partial path of length d at simulation k has path reward equal to $Q(s, a')$, i.e. $\rho_k(d) = Q(s, a')$. In the next simulation i.e. $k + 1$, the MCRT-CAS technique can have one of two case. In the first, if $Q(s, a)$ has not converged in t rollouts, then $Q(s, a')$ either remains the same or increases (due to the monotonic property of the action value function). Therefore, Proposition 2 is true for $k+1$ in the first case. In the second case, if $Q(s, a')$ has converged in t simulations, then the MCRT-CAS technique samples another action $a_r \in A(s)$ that is the second best and computes $Q(s, a_r)$. If $Q(s, a_r) < Q(s, a')$, again Proposition 2 is true for $k + 1$. However, if $Q(s, a_r)$ is greater than $Q(s, a')$ then the MCRT-CAS technique updates the definition of the best action at s. In this scenario, a_r becomes the best action at s. This also implies

that the MCRT-CAS technique discovers a new partial path of length D from s with a higher reward value than the previous best partial path i.e. $\rho_{k+1}(d) > \rho_k(d)$. In this case, $s_r = T(s, a_r)$ is an element of the new shortest path from s and a_r is the best action. This makes Proposition 2 true in case of the discovery of a new best action at s. Hence, Proposition 2 is true for $t = k + 1$ simulations.$\square$

4.2 Multi-Goal MOCART (MG-MOCART) Planner)

The MG-MOCART planner is designed to solve a path planning problem that has multiple goals. For example, the planning agent may get a path planning problem with the following goals:

1. Move from its current position to a mineral mine.

2. Once it reaches the mine and pick up some minerals, then move to a base centre.

3. Move towards the mineral mine after storing the mineral in the base centre.

The planning agent keeps moving to different goal locations until the termination condition is true. This kind of path planning is different from the commonly known path planning problem that has only one initial state and one goal state. The path planning problem with multiple goals is often required in computer games (such as Real-Time Strategy games) to complete a task, e.g. resource collection.

4.2.1 Overview

The MG-MOCART Planner interleaves planning and plan execution like the MO-CART planner. It modifies the MOCART planner with two new features. First, it builds a tree (called a Rapid Tree or RT) of the states that the planning agent has been occupying since the start of the planning process. The details of RT are given in section 4.2.3.1. The second feature of the MG-MOCART planner is the use of two stages to solve planning problems with multiple goals. In the first stage, the MG-MOCART planner runs the MCRT technique to estimate the action values and to choose an action at the current state to reach a given goal state. In the second stage, the MG-MOCART planner exploits RT to plan an action to reach the goal state.

4.2.2 Intuition

In a planning problem with multiple goals, it is intuitive to reuse the information gathered from the previous searching efforts. The design of the MG-MOCART planner is based on this intuition. The MG-MOCART planner reuses the previously computed estimates of the action values to achieve similar goals. It builds a tree of the states that the planning agent visits to reach the first goal state and reuses it to reach other goal states if the goal states are already in the tree. The tree (RT) can be thought of as a small part of the search space that can be exploited to achieve fast action selection at an already seen state. RT is built and updated during online planning. The MG-MOCART planner uses RT once the first goal is reached and it is updated if any state on RT is occupied by a static obstacle. The MG-MOCART

planner is designed as a two-stage planner; one stage uses the MCRT technique and the other uses an informed breadth first search (on RT) to choose actions to solve path planning problems. The planner can shift from one stage to another depending on some conditions (e.g. if the first goal is achieved then start using the tree).

4.2.3 Algorithmic Details

In this section we describe how to embed the MCRT technique into a real-time path planner with a list of goal states to visit. The planner is supplied with the sensor information described in the problem definition (section 1.4 of Chapter 1), and interleaves planning and execution as follows. The algorithmic details of the MG-MOCART planner are shown in Figure 4.2. The algorithmic details use the notation defined in section 3.2 of Chapter 3. Lines 1 and 2 (Figure 4.2) initialise the tree (RT). At the start of the planning loop in line 4, "pop g from G" has the effect of assigning g to the head of list G, and reducing G. If G is found to be empty, then the "pop" function will exit the loop and the program will end. The planner proceeds in two stages: in the first stage (lines 5 - 10), RT is populated using the MCRT technique, interleaved with action execution (line 9). RT builds up a tree of collision free states of the search space within S, as it is possible that s can be revisited if the states are retraced or $Execute(a, s)$ leaves s unchanged - where there is an obstacle in front of s in the direction of a. RT is expanded with an edge from s to s_{next} if it passes the validity test: this test returns true unless s_{next} already appears in RT, or if s already has a child edge in RT that has a greater estimated reward than s_{next}. In either case, the returned action is executed on s and the new state of the agent recorded. After

Procedure *MG-MOCARTPlanner*

Read access MDP formulation (S, A, P, R, s_o, G);

1.   initialise tree $RT := null$ and state $s := s_o$;

2. $RT.AddRoot(s)$;

3.   WHILE G is not empty

4.   pop g from G;

5.   REPEAT

6. $a := MCRT(s, g, d, n)$;

7. $s_{next} := T(s, a)$;

8.   IF $Valid(s_{next}, s, RT)$ THEN $RT.AddNode(s, s_{next})$;

9. $s := Execute(a, s)$

10.   UNTIL $s.pos = g$;

11.   pop g from G;

12.   REPEAT

13. $a := LocalPlanner(s, g, RT)$;

14. $s_{next} := T(s, a)$;

15.   IF ObstacleFree(s_{next})

16.   THEN $s := Execute(a, s)$

17.   ELSE $RT.Remove(s_{next})$;

18.   IF $s.pos = g$ THEN pop g from G

19.   UNTIL $\neg$ (ObstacleFree(s_{next}) AND g is on RT)

20.   END WHILE

End *MG-MOCARTPlanner*

Figure 4.2: The MG-MOCART planner.

the first stage achieves the first goal, the planner enters the second stage. In the second stage (lines 12 - 19), the planner exploits the fact that RT has been built up in

the first phase, searching RT to find the path to the next goal state from the current location. The local planning method (line 13, Figure 4.2) is a breadth first search of fixed depth to find an action to the neighbouring state of the current state which reduces the distance to the next goal state. If the simulation of action a (utilizing the stochastic transition function T) changes the agent's position to a position which is obstacle free, then that action is actually executed, otherwise the state is removed from the tree. The planner leaves the second stage and starts running the first stage again if s is occupied by an obstacle. This interchange between stage one and two continues until the end of the game when all goal states have been reached, or a fixed time bound is reached for the game.

4.2.3.1 Local Planner and Rapid Tree

RT is a collection of the states that the planning agent visits to reach the first goal state in a multiple journey planning problem. The states, in RT, are connected via edges. RT is stored as a tree. In the absence of cycles and dynamic obstacles in the domain world, RT can be in the form of a chain of nodes such that each node has only one child node. However, this hypothetical case is not true for RTS games. The addition of new nodes or edges in RT requires two conditions to be true. The first condition is to verify that the corresponding edge or node is not already in the tree. The second condition checks whether the addition of a new edge between two given nodes does not create a cycle in the tree. In the case of a cycle, an edge with a smaller reward value is removed from the tree. These checks are made by the function $Valid$ (line 8, Figure 4.2).

$LocalPlanner(s, g, RT)$ traverses RT (for action selection at s to reach the current

goal g). The planner performs the look-ahead search in two steps. In the first step, it checks whether s is a part of the tree. If s is a part of RT, then the local planner finds the child nodes and the parent node of s and stores them in a priority queue ordered by the heuristic values. In the second step, the planner selects the top node of the queue as the best neighbouring state (of s). Then the local planner chooses an action at s - using the function $SelectAction$ of MCRT (line 3, Figure 3.4) - to move the planning agent to the best neighbouring state. If s is not in RT, then the MG-MOCART planner switches to the first stage.

4.2.4 Completeness

The MG-MOCART planner is an extension of the MOCART planner. It is based on two stages which uses local information around the current state of the planning agent to plan an action for execution. The first stage uses the MOCART planner and builds an an RT tree. The second stage uses a local path planner that finds an action by searching the tree built in the first stage. The planner terminates after a fixed time duration. The first stage of the planner is exactly like the MOCART planner except for the building up of a tree based on the output of the planner. The MOCART planner can eventually reach a goal location, therefore, the first stage of the MG-MOCART planner ensures completeness. In the second stage, the MG-MOCART planner uses a local search planner to choose an action at the current state of the domain world. The local search planner does not modify the action values but it imposes a limit on the number of times an action is executed at a state. If g is on the tree, then the MG-MOCART planner will eventually reach the goal state. If g

is not on the tree then the MG-MOCART planner shifts to the first stage and uses the MOCART planner to choose the actions to reach g and extends the tree. The MG-MOCART planner always reaches a goal (given in the goal list) using either the first stage or second.

4.2.5   Correctness

In the first stage of the MG-MOCART planner, the selection of an action at the current state of the planner is performed by the MOCART planner. According to Theorem 4, the MOCART planner always selects the best action ($a \in A(s)$) at the current state (s) of the planning agent such that a leads to the best successor state ($s' \in Succ(s)$) of s. The best successor state has the shortest distance to the goal state with respect other successor states. As the first stage of MG-MOCART planner is a MOCART planner, therefore, MG-MOCART planner also selects the best action at the current state of the planning agent to move the planning agent along a shortest path relative to the neighbourhood of the current state.

The second stage of the MG-MOCART planner is based on a tree built during the first stage of the planner. This stage reuses the outcomes of the first stage. The second stage (i.e. *LocalPlanner*) selects the best child state of the current state to move towards the goal. Therefore, the best action selected by the second stage at the current state (s) moves the planning agent on the shortest path to the goal location according to the current neighbourhood of the planning agent. The same selection was made by the MOCART planner when the same state was seen in the first stage (as each node in the tree is an outcome of the MOCART planner). Therefore, Theorem

4 is also true for the MG-MOCART planner.

4.3 Summary

This chapter described the details of two kinds of real-time path planners that are based on Monte-Carlo simulation techniques: MCRT and its variations. The algorithmic details along with the correctness and completeness properties were described for each planner. Three Monte-Carlo path planners - that belong to the first category of planners - are MOCART, MOCART-RAS and MOCART-CAS. These planners can solve path planning problems that have only one goal. These planners interleave planning and plan execution to solve a path planning problem under real-time constraints and partial visibility. These planners also update the state transition probabilities during online planning. These planners select an action that is best according to local information and always reach the goal position if there exists a path from the start location to the goal position. The completeness of the MOCART, MOCART-RAS and MOCART-CAS planners was also discussed to highlight the cases when these planners always reach the goal state for a given path planning problem. The correctness of MOCART and MOCART-CAS planners was also described in this chapter. In the second category, a planner was presented in this chapter that can solve planning problems with multiple goals. This planner is called MG-MOCART planner which is an extension of the MOCART planner. The MG-MOCART planner works in two stages where one stage is based on the MOCART planner and the other reuses the outcomes of the first stage. The algorithmic details of the MG-MOCART planner were given along with the correctness and completeness of the planner. To investi-

gate the quality of the solutions by each planner, we run these planners on different path planning problems from the path planning benchmarks. The details of these experiments are given in the next chapter.

Chapter 5

Results and Analysis

This chapter provides details about the experimental works which have been carried out to measure the performance of the Monte-Carlo real-time path planners and compare their performance against their closest competitors. We have designed two sets of experiments to test the Monte-Carlo planners. The first set of experiments uses the pathfinding benchmarks (available at *http://www.movingai.com/benchmarks/*) to evaluate three Monte-Carlo planners (MOCART, MOCART-RAS and MOCART-CAS). Nathan Sturtevant and BioWare made these benchmarks available for research. These planners solve planning problems with a single goal. The second set of experiments uses a typical RTS game to test MG-MOCART planner (section 4.2, Chapter 4) on three hand-crafted game maps and each planning problem in this set of experiments has more than one goal. The details of the experimental design, results and their analysis in the first set of experiments are given in section 5.1. Section 5.2 is about the second set of experiments performed to test MG-MOCART planner in a RTS game and analyses the results of these experiments.

The key performance indicator for all the planners is the fast action selection mechanism. In the first set of experiments, fast action selection is represented by time per search. Smaller time per search represents better planner performance (in terms of fast action selection). In the second set of experiments, the score of a planner indicates performance of a planner with respect to fast action selection. A planner with the highest score is better than others in the context of fast action selection.

5.1 Experiments on Benchmarks

These experiments are designed using four benchmarks. The details of these benchmarks are given in section 5.1.1. From each benchmark, a set of planning problems is selected for an experiment. A planner is run to solve all these planning problems on a benchmark in an experiment. Three performance measurement parameters are used as an outcome (or result) of an experiment. The details of the performance parameters are given in section 5.1.1.3. These results are then used to compare the performance of Monte-Carlo planners (i.e. MOCART, MOCART-RAS, MOCART-CAS) with their closest competitors. These planners have been defined in section 4.1 of Chapter 4. We selected two closest competitors to compare with these planners. Section 5.1.1.1 provides the motivations for selection of these rival planners to compare the results with Monte-Carlo planners. To compare the results, we perform several experiments for each planner on a benchmark. Each experiment is carefully designed using a combination of the algorithmic parameters of each planner. The details of the algorithmic parameters of each planner (along with the carefully chosen range of their values) are described in section 5.1.2. The experiments are performed

for both static and dynamic worlds of each benchmark. In static worlds, the original topology of the map (of the corresponding benchmark) remains the same for an experiment. In dynamic worlds, the topology of the world is changed by the creation of new obstacles during an experiment. The new obstacles are created randomly (by the research tool) in each problem of the benchmark in an experiment. The results and the analysis of the results in the static world of the benchmark are given in section 5.1.3 while section 5.1.4 describes the results of the experiments in the dynamic worlds of the four benchmarks. All experiments on the benchmarks (both in static and dynamic worlds) are performed using the MAI tool. Details about the tool are given in 5.1.1.2.

5.1.1 Benchmark Selection

To investigate the performance of the planners, we selected four maps from the commercial game benchmarks. These benchmark maps are from a commercial game called *Dragon Age: Origins* (this game was published in 2009). The details about the name, size and number of planning problems of each benchmark map are given in Table 5.1. Three of these maps (i.e. Orz103d, Orz702d and Orz900d) have been explored in the real-time path planning context in a recent work [31]. Each benchmark map has a scenario file associated with it. The scenario file of a map contains several path planning problems. A planning problem in a scenario file of a map has the coordinates of the initial state and a goal state on that map along with the optimal path length for this problem. We sequentially select planning problems from each benchmark, starting from the first planning problem up to a fixed number. For example, in Arena2,

the first 300 planning problems are chosen for the experiments. It is notable from Table 5.1 that the selected benchmark set has a large variety of planning problems ranging from a very short path length (i.e. 1) to much larger (i.e. 179.95) on four different maps. The size of the set of the planning problems is 1350. Every problem has a different start location and goal position from the others. The size of the maps also range from 281×209 cells to 1491×696 cells. All maps are of octile type. Octile type means the map is a grid where the cost of an edge to an adjacent cell of the grid is 1 and the cost of a diagonal edge is $\sqrt{2}$. The total initial heuristic error (E) [49] of a benchmark is used as a hardness measure for the benchmark. The total initial heuristic error (E) is computed using equation (5.1) where $h^*(P_i)$ is the optimal path length of planning problem P_i, $h(P_i)$ is the initial heuristic value of P_i for all $i = 1, 2, ...n$. n is the total number of planning problems. The initial heuristic value of a planning problem is computed by using the octile heuristic between the start state and the goal state of the problem. A benchmark map is more challenging than another benchmark if its total initial heuristic error is higher. The obstacle ratio [49] is the second parameter that has been used to represent how challenging a benchmark map is. The obstacle ratio r is computed by using the ratio of the number of obstacles to the size of the map i.e. $\frac{R}{width\times height}$ where R is the number of blocked cells in a map of size $width \times height$. The obstacle ratio of each benchmark map is shown in Table 5.1. The real-time heuristic search planners, e.g. LRTA, have been shown to be sensitive to the number of obstacles in a map and perform poorly when there are a large number of obstacles [60][49].

$$E = \sum_{i=1}^{n} |h^*(P_i) - h(P_i)| \tag{5.1}$$

120

Map	Size	No of problems	Total Initial Heuristic Error	Longest Path	Obstacle Ratio(%)
Arena2	281 × 209	300	2672.26	119.80	58.60
Orz103d	463 × 4456	300	2502.39	119.70	80.67
Orz702d	718 × 939	450	1914.93	179.95	89.42
Orz900d	1491 × 656	300	1073.71	119.28	90.12

Table 5.1: The selected benchmarks.

The Arena2 map is of smaller size than the other benchmark maps but it has a higher total initial heuristic error than the other benchmark maps. Therefore, it is more challenging than other benchmarks. The Arena2 map is shown in Figure 5.1. The black shaded area represents an obstacle (or impassable area) and the white shaded area (except the text) is empty (or passable). The agent can move in the empty area only. The topology of the map consists of several thin (and small size) obstacles and two large walls. The main challenging areas are the walls and a small enclosed area which has an extremely narrow passage to link with the rest of the map. The narrow passage and a wall are highlighted in the figure. Figure 5.2 shows the Orz103d map. The definition of empty spaces and static obstacles is the same as given in Arena2. The topology of the map consists of small regions which are connected via (narrow and wide) streets. Some of the regions are very close to each other but they are not directly connected (for example two regions highlighted in Figure 5.2). This makes path planning challenging because the admissible heuristics (e.g. octile heuristic) can misguide the local search path planners to move to a region that is very close to the goal location but it has no (passable) link with a region of
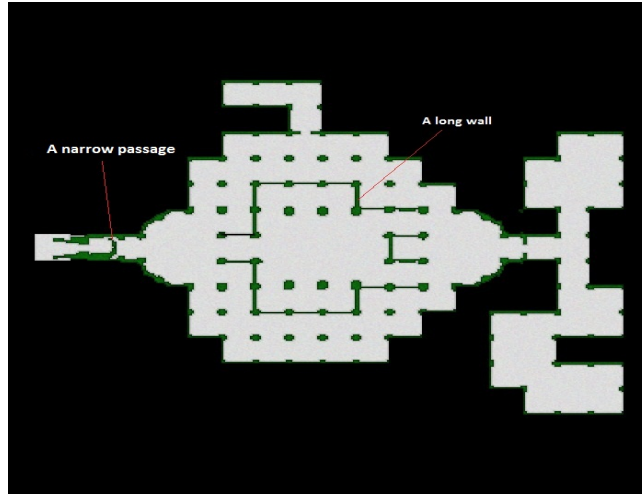
Figure 5.1: Arena2 map.

the goal location. The Orz103d map is less challenging than Arena2 with respect to total initial heuristic error. The Orz702d map is shown in Figure 5.3. The empty
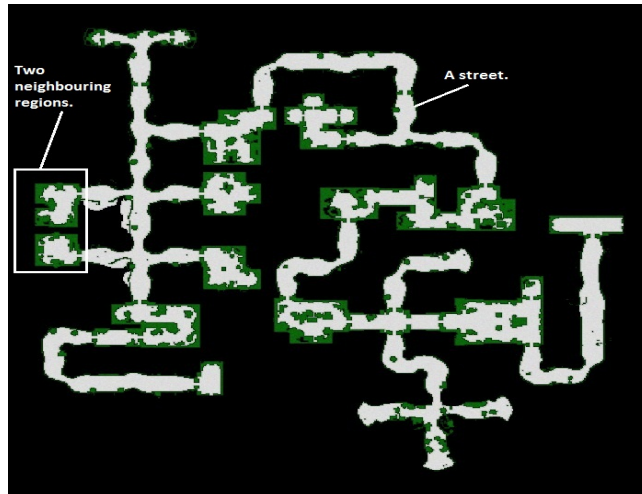

Figure 5.2: Orz103d map.

area is white and the obstacles are in black. This map has three big empty regions (as shown in Figure 5.3) which are aligned diagonally and linked via narrow passages.

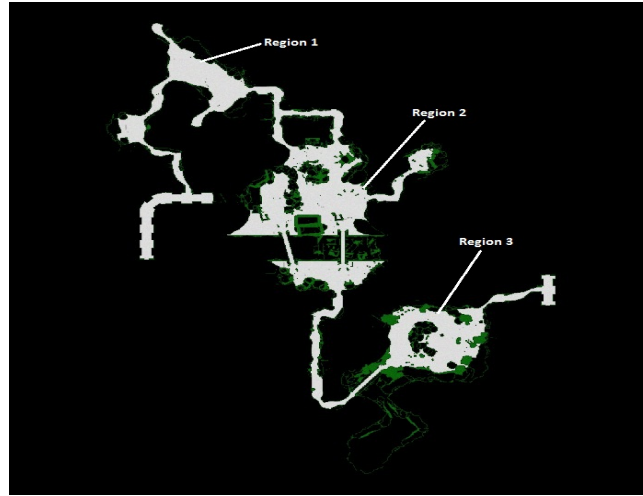This map is less difficult than Orz103d (and Arena2) with respect to the total initial heuristic error.

Figure 5.3: Orz702d map.

Figure 5.4 shows the image of the Orz900d map. The topology of this map is the most challenging for two reasons. (1) The size of map is very large; (2) it has several regions which are close to each other but not connected directly with each other. The geographical neighbouring regions have connections through long, irregular and narrow passages. A shortest distance heuristic can easily misguide a local search planner to get stuck in a large region that has no link with the passage that goes to the goal location.

5.1.1.1 The Closest Competitors

The performance of the real-time Monte-Carlo path planners is compared against two recent real-time path planners: Local Search Space-Learning Realtime A* (LSS-LRTA) and Real-time D*-Lite (RTD). These planners are based on real-time heuristic
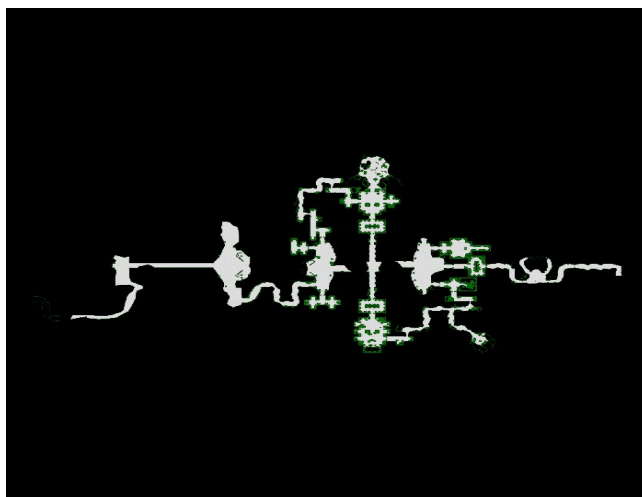
Figure 5.4: Orz900d map.

search and incremental heuristic search. These planners can solve a path planning problem under real-time constraints with incomplete information about the domain world. The details of these planners (and their suitability for the real-time path planning problem in RTS games) have been discussed in section 2.1.4 Chapter 2.

5.1.1.2 The Mokon Artificial Intelligence (MAI) Tool

The MAI tool, developed by David Bond, is an easy to use tool for researching path planning problems under partial visibility and real-time constraints. The tool is built using C# and Microsoft Visual Studio 2010. The main motivations for using this tool are as follows:

(1) The tool has built-in algorithms (like LSS-LRTA and RTD) which we want to compare with the real-time Monte-Carlo path planners.

(2) It also provides a library to load the benchmark maps and generate a grid world for the benchmark maps. It allows the creation of static and dynamic grid worlds for

a benchmark map.

(3) It provides a platform to run the path planners under tight time constraints and partial visibility.

The scenario file and the map file of each benchmark map are in an ASCII format, therefore, we use the *StreamReader* class (Microsoft .Net Framework 4.0) to read these files. Each line of a scenario file is a path planning problem. The program reads the map size, path of the map file, the start location, the goal location and the length of the optimal solution path of each problem given in the scenario file. A grid world of the benchmark map is generated for each planning problem by using the *HOGGridWorldLoader* class (which is part of the tool) because the planning problems in a scenario file can have different sizes of map for the same benchmark. If the problem has a different grid world size to the actual size of the benchmark map, then the tool can re-scale it to the size given in the problem definition. A path planner is run on the generated grid world for the given path planning problem by using the *run* function of the *AlgorithmRunner* class of the tool. The tool imposes two kinds of limits on the planner for a given planning problem. The first limit is on the planning time per planning episode. This is called a computational limit. The second limit is on the length of solution. This is called *pathlimit*. *pathlimit* is compared against the ratio of the current length of the path traversed by the planner on the map to the optimal length given in the problem definition. If this ratio is greater than *pathlimit* at any time during planning, the problem is declared as unsolved by the planner.

The MAI tool encodes the actions as a navigational compass. The action set is: { MOVE TO NORTH, MOVE TO EAST, MOVE TO SOUTH, MOVE TO WEST, MOVE TO NORTH EAST, MOVE TO SOUTH EAST, MOVE TO SOUTH WEST,

MOVE TO NORTH WEST, NO MOVE}. $NOMOVE$ action does not change the position of the planning agent. It is used in a dynamic world in a case when there is no valid move for the agent.

5.1.1.3 Performance Measurement

The performance of a planner is measured using three parameters: time to solve a set of planning problems (given in Table 5.1) for a benchmark map, sub-optimality of the solution by the planner and time per search. The total time to solve is represented by T_p in the rest of the thesis. For example, T_p for an experiment on $Arena2$ means the total time of search (in milliseconds) by the planner to solve 300 planning problems. Time per search means the time taken by an algorithm to select one action in one planning episode. Time per search is represented by T_s in the rest of the thesis. Sub-optimality is measured using a ratio of l_p to l_o (i.e. $\frac{l_p}{l_o}$) where l_p is the length of the solution by the planner and l_o is the length of the optimal path given in the benchmark. Sub-optimality is represented by Sub. Time per search is the key performance indicator as it is important for a real-time system to respond within the shortest time interval. Smaller values of this parameter represent better planner performance. Sub-optimality of a planner can not be smaller than one as no planner can find a path of length shorter than the optimal path length. A planner has a good performance if its sub-optimality is close to one.

If a planner cannot solve all planning problems (selected for a benchmark), then we use the scale parameter. A scale of a planner is the number of planning problems solved by the planner out of the total problems selected for the benchmark. For example, a planner has a scale of 250 on Arena2 if it solves 250 problems out of 300

problems (selected for Arena2 benchmark).

5.1.1.4 Implementation

All planners, i.e. the Monte-Carlo real-time planners and the competitors (LSS-LRTA and RTD) are programmed in C#. LSS-LRTA and RTD are available in the MAI tool while the Monte-Carlo planners are programmed by the author of this thesis. In the Monte-Carlo planners, the state transition probabilities, the action values (i.e. $Q(s, a)$ for every a applicable at state s) and the statistics (e.g. number of times an action is sampled at a state) are stored using hash tables. Initially, all these tables are empty, they are populated if a new state is seen by the simulation model.

5.1.2 Algorithmic Parameters

We designed a set of experiments based on the algorithmic parameters of each planner. All real-time Monte-Carlo planners have four common parameters which are n_c (the convergence parameter), w_d (the weight factor), $timelimit$ (number of simulations) and $Depth$ (look-ahead search depth). The carefully designed set of values for three parameters are shown in Table 5.2 while w_d is set to one i.e. $w_d = 1$. The main reason to select this value of the weight factor is to keep the heuristic admissible (as the rival techniques use an admissible heuristic). All planners use the same heuristic (i.e. octile heuristic) as a shortest distance estimate. The octile heuristic is a commonly used heuristic in octile grids (e.g. [31], [11] and [41]). The selection of the values of the other three algorithmic parameters (i.e. $timelimit$, n_c and $Depth$) is based on the initial experiments for the real-time Monte-Carlo path planners. We noticed from the initial experiments that these planners can perform well with even smaller

n_c (Convergence)	$timelimit$	$Depth$
1	1	1
3	10	3
5	30	6
10	60	9
15	90	12
20	120	15
30	200	15
500	-	-
1500	-	-

Table 5.2: Algorithmic parameters and their values for the experiments.

values of $timelimit$ than 200 rollouts and higher values of $timelimit$ than 200 rollouts is expensive in terms of time per search. RTD has two algorithmic parameters. These two parameters are $LocalRatio$ and the computational limit (or depth of the look-ahead search). The details of these parameters are in section 2.1.4 of Chapter 2. LSS-LRTA has one algorithmic parameter - the computational limit (or depth of the look-ahead search). We explored RTD on the benchmark problems with six values of $localRatio$ which are {10, 20, 30, 40, 50, 75}. The selection of this range of values for $locationRatio$ is influenced by the results of RTD presented in [11]. The maximum computational limit for all planners i.e. MOCART, MOCART-RAS, MOCART-CAS, RTD and LSS-LRTA is the same i.e. 15. The reason for the selection of this value is to keep the look-ahead search within the visibility range. Higher values of the look-ahead depth are also not fair for the experiments as local search of depth greater

than the length of a planning problem becomes a global search.

5.1.3 Results and Analysis - Static World

The experiments are performed on seventeen different machines with the same hardware and software configurations. Each machine has Intel(R) Core (TM) 2 Quad processors each of speed 2.6 GHz CPU speed and 8 GB RAM. These computers use Microsoft Windows 7 as an operating system. The results given in this section are computed by using $pathlimit = 100$. This termination condition is selected after several initial experiments with different values of $pathlimit$ on three benchmarks (Arena2, Orz103d and Orz702d). With this limit, at least one planner solves all planning problems in three benchmarks. The results in the static world of each benchmark map are discussed in the following sections.

5.1.3.1 Arena2

The best results of each planner on the static world of the Arena2 map are shown in Table 5.3. RTD solves all planning problems. LSS-LRTA, MOCART and variations of MOCART are not scalable in this benchmark. LSS-LRTA is more scalable than MOCART and its variations. MOCART-CAS performs significantly better than all other planners with respect to time per search. This shows the MOCART-CAS planner has the fastest action selection mechanism as compared to the other planners. The better performance of MOCART-CAS is due to the use of the corridors of the actions. To discuss the sub-optimality of the Monte-Carlo planners on Arena2, we selected a problem from Arena2 as an example. We call this problem $P1$. The initial and the goal locations of this problem are shown on the Arena2 map in Figure 5.5.

129

Planner	$Depth$	$timelimit$	n_c	Avg T_s	Avg Sub	Scale
MOCART	15	200	500	0.0172	11.54	273/300
MOCART-RAS	15	200	500	0.005	12.34	271/300
MOCART-CAS	3	30	1500	**0.0004**	7.04	284/300
RTD	15	-	-	0.083	3.63	300/300
LSS-LRTA	15	-	-	0.002	2.96	296/300

Table 5.3: Time per search, Sub-optimality and Scale of five planners on Arena2 (Avg means Average).

There is a long wall between the initial and goal locations. The wall is not straight
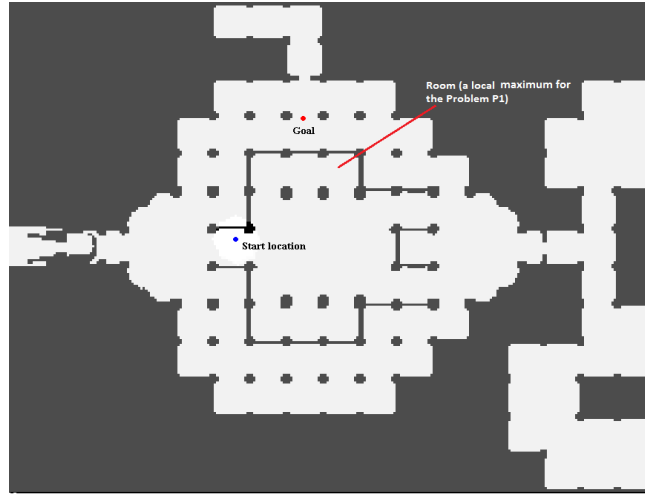

Figure 5.5: The planning problem $P1$ on Arena2 map.

and creates a large room of a rectangular shape as shown in Figure 5.5. The problem is challenging for any local search planner based on admissible heuristic search because the admissible heuristic prefers the path towards the large room and there is no path to the goal location via this room. The room can be considered as a local

maximum (or minimum) for the local search planners.

The path found by MOCART for this problem is shown in Figure 5.6. MOCART keeps moving inside the large room for a long time. The planner visits every empty
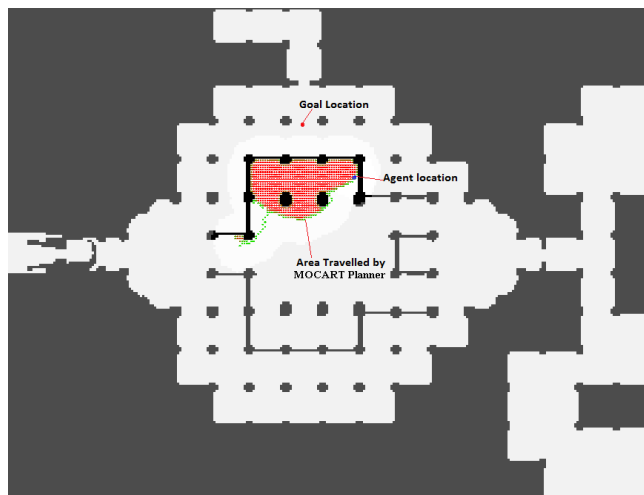

Figure 5.6: The path followed by MOCART to solve $P1$ on Arena2.

space in this room for a few times and then it learns to avoid moving inside the room. Learning to avoid moving towards a previously visited state is guaranteed by a limit on the number of times an action is executed at a state. The process of learning to escape the local maximum is very slow in MOCART because this process depends on the limit on the action execution at a state and every state in the big room has more than one applicable action such that execution of any one of these actions keeps the planning agent moving inside the room (local maximum).

The solution of MOCART-CAS for the same problem is shown in Figure 5.7. MOCART-CAS follows the same path (as that followed by MOCART) to solve $P1$ but it learns quicker than MOCART to get out of local maximum. MOCART-CAS learns quicker
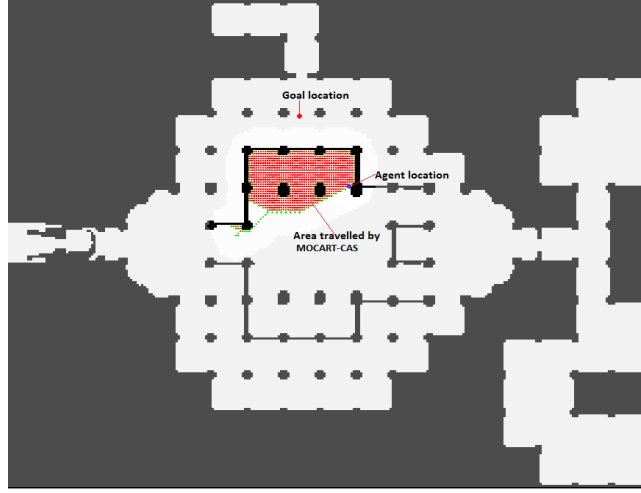
Figure 5.7: The path followed by MOCART-CAS to solve $P1$ on Arena2.

(to escape the local maximum) than MOCART because of the use of the corridor of actions in a rollout. In MOCART-CAS, if an action at a state has exceeded the limit of execution then it is excluded from the corridor and hence it is not sampled in the look-ahead search. But the look-ahead search in MOCART does not impose this restriction on an action. Therefore, MOCART-CAS has fewer actions which are executed for the given limit and it learns more quickly to avoid moving to the same state again.

The path found by LSS-LRTA to solve $P1$ is shown in Figure 5.8. LSS-LRTA could not reach the goal location within *pathlimit*. By comparing the performance of LSS-LRTA with MOCART (Figure 5.6) and MOCART-CAS (Figure 5.7) for the same problem i.e. $P1$, it is notable that LSS-LRTA also keeps traveling in the big room for a long time but it also moves to areas other than the room (for example towards the second wall of the Arena2 map). It escapes the local minimum (i.e. room) for a few times but again moves into the room. LSS-LRTA learns to avoid moving in
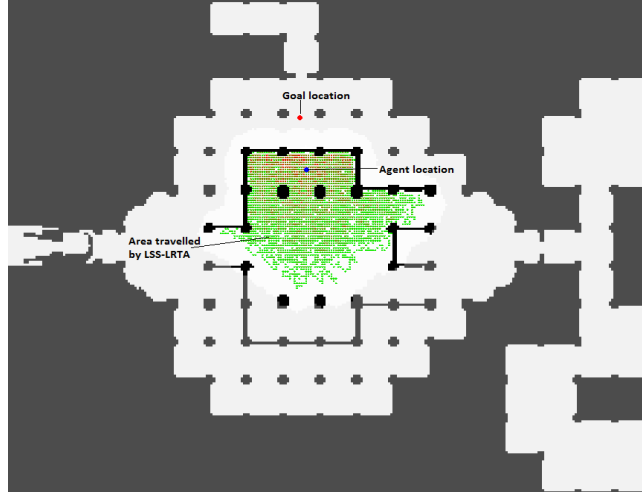
Figure 5.8: Path followed by LSS-LRTA for the planning problem $P1$.

the room by updating the heuristic values of the states it sees during the look-ahead search. However, if the change in the heuristic values (of the locations inside the room) are inaccurate with respect to the heuristic values of the neighbouring locations (outside the room), it can again move inside the room (even after moving out of the room). The inaccurate heuristic values create the heuristic depression (also defined and discussed in section 2.1.4 of Chapter 2) for LSS-LRTA. The area inside the room is a heuristically depressed region for LSS-LRTA. It can be deduced that the poor performance of LSS-LRTA in Arena2 is due to the presence of heuristically depressed areas in those planning problems which are unsolved by LSS-LRTA.

RTD is efficient in Arena2 with respect to sub-optimality but it is still poor in terms of time per search. The RTD planner successfully solves all planning problems on this map under the given *pathlimit*. The problem $P1$ is also used to analyse the performance of RTD with respect to sub-optimality on Arena2 more closely. The path followed by RTD to solve $P1$ problem is shown in Figure 5.9. RTD explores
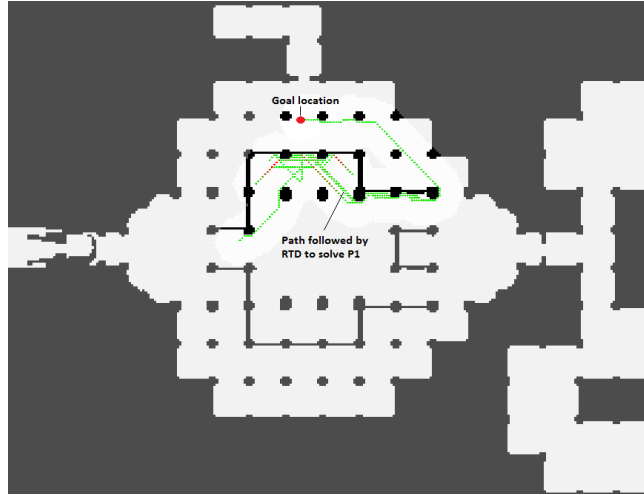
Figure 5.9: The solution of $P1$ by RTD.

the boundaries of the wall to find the exit from the local minimum and eventually finds a way to move to the goal location. This success of RTD is due to the global search i.e. D*-Lite. The backward search guides the planning agent to move towards the corners of the walls. RTD does not behave like LSS-LRTA or MOCART (and its variations) because the other side of wall was visible to the global search algorithm. This example highlights the importance of the use of the global backward search in combination with a local forward search algorithm to solve path planning problems in a partially visible grid world. The Arena2 results show that RTD is not sensitive to the initial heuristic error and it performs well if the obstacle ratio is small.

5.1.3.2   Orz900d

The results of the five planners on Orz900d are shown in Table 5.4. There is no single planner that can solve all planning problems on Orz900d. The results show that MOCART-CAS performs better than all other planners with respect to time per

Planner	$Depth$	$timelimit$	n_c	Avg T_s	Avg Sub	Scale
MOCART	15	30	15	0.0073	7.46	287/300
MOCART-RAS	15	30	500	0.0046	7.34	286/300
MOCART-CAS	3	30	6	**0.0004**	3.84	293/300
RTD	15	-	-	0.0138	3.89	292/300
LSS-LRTA	15	-	-	0.002	1.46	299/300

Table 5.4: Scale of five planners on Orz900d.

search. The MOCART-CAS planner performs better than MOCART and MOCART-RAS in terms of scalability. It has the same scalability as RTD. MOCART and MOCART-RAS solved the same number of problems (this is due to the large similarity between both algorithms in terms of action sampling). For a detailed analysis of these results with respect to sub-optimality, we choose a problem from Orz900d as an example. This problem is named $P2$. The initial and the goal locations of this problem are shown in Figure 5.10. The challenging issue in this problem (i.e., $P2$) is the presence of two local maxima (or minima). The first local maximum is a concave region near the start location. This region is shown as a gulf in Figure 5.10. The second local maximum is a narrow passage that leads to a dead-end. The narrow passage is highlighted as a second local maximum in Figure 5.10. The path to the goal location passes via a wide passage which is on the opposite side the local maxima.

The path followed by MOCART to solve $P2$ is shown in Figure 5.11. The figure shows that MOCART is stuck in the first local maximum (i.e. the gulf). The MOCART planner learns gradually to move out of the gulf and starts moving towards the goal
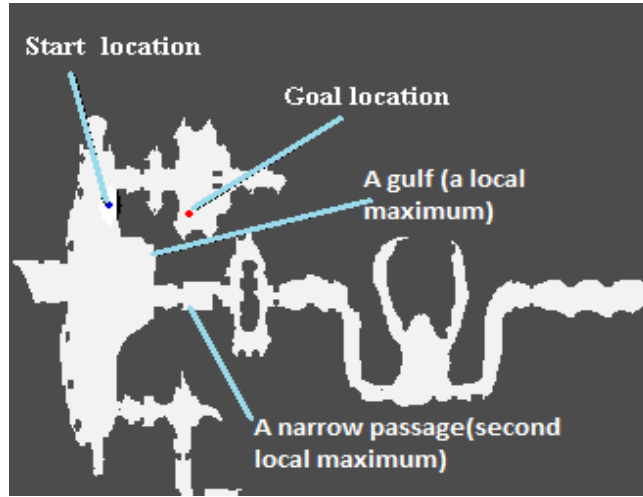
Figure 5.10: The start and goal locations of $P2$ planning problem on the Orz900d map.
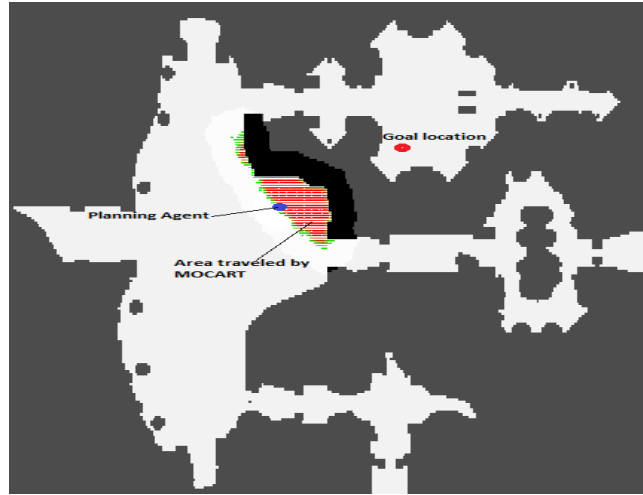

Figure 5.11: The path followed by MOCART to solve $P2$.

location in the right direction. MOCART does not move to the second local maximum after escaping the gulf. The MOCART planner avoids moving towards the second local maximum mainly due to the unchanged heuristic values of the states. The look-ahead search of the MOCART planner always samples the actions towards the

locations inside the gulf (because these locations have the minimum aerial distance to the goal location according to local information). But the planning agent can not move deep inside the gulf for more than a certain number of times as the planner imposes a limit on the number of times an action is executed. As a compromise between the limit on action execution and the heuristic based action preferences, the MOCART planner keeps moving in the places near the first local maximum rather than moving in the narrow passage (or the second local maximum). Even the MOCART planner does not solve the problem within the given *pathlimit* but it performs well in avoiding the second local maximum.

Figure 5.12 shows the path followed by the MOCART-CAS planner to solve $P2$. The planner moves to a larger area than MOCART. That is, the MOCART-CAS
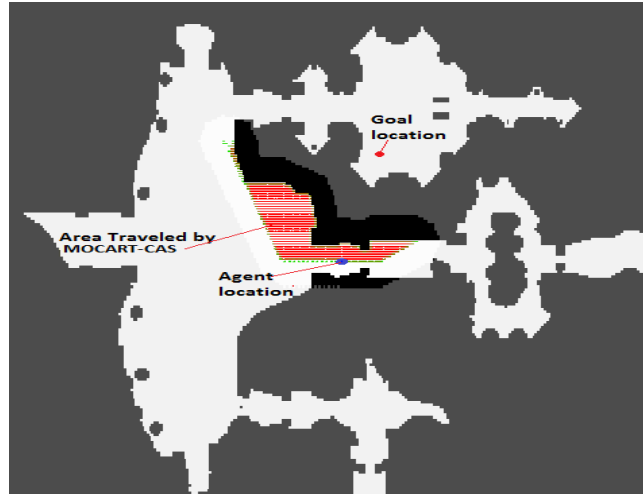

Figure 5.12: Path followed by MOCART-CAS to solve $P2$.

planner moves out of the gulf and then enters into the narrow passage (the second local maximum). However, the MOCART-CAS planner remains very close to the

entrance of the narrow passage. The MOCART-CAS planner appears quicker than the MOCART planner in terms of escaping the first local maximum. This is due to the use of corridors in the MOCART planner. If an action exceeds the limit of execution, its corridor is not sampled in the look-ahead search. This property of the MOCART-CAS also makes it to move to the second local maximum, however, the planner also learns quickly to escape the second local maximum. The MOCART-CAS planner keeps moving around the first local maximum due to the use of raw (or unchanged) heuristic in the rollouts.

The path followed by LSS-LRTA is shown in Figure 5.13. LSS-LRTA also explores a large area around the first local minima and moves a long way inside the narrow passage. However, LSS-LRTA eventually manages to escape both first local minima
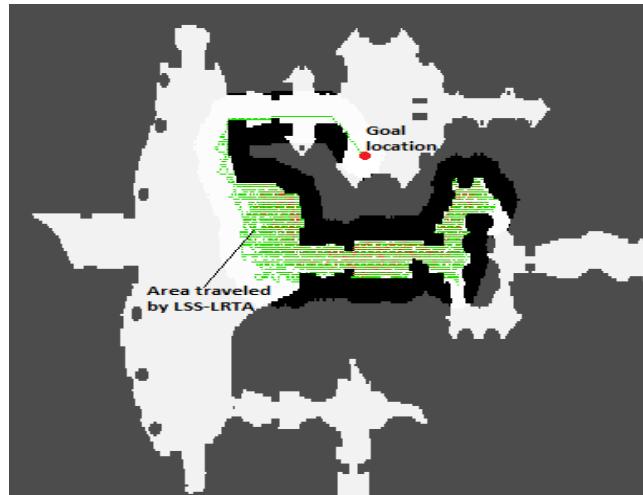

Figure 5.13: Solution of $P2$ by LSS-LRTA.

and the second one within *pathlimit* to solve $P2$. MOCART-CAS also explores the same area but with a slow speed and does not move as far in the wrong direction

(inside the narrow passage) as LSS-LRTA does. A comparison of Figures 5.12 and 5.13 reveals that the recovery of MOCART-CAS from the initial error (in the selection of the right action) is slower than LSS-LRTA. The reason behind the slow recovery of the MOCART and MOCART-CAS planners from the initial errors is that these Monte-Carlo techniques learn (or update) the action values of one state in a planning episode while LSS-LRTA updates the heuristic values of all states seen in the look-ahead search in a planning episode.

Figure 5.14 shows the path followed by RTD to solve $P2$. RTD moves further inside the narrow passage than LSS-LRTA, MOCART and MOCART-CAS. Figure 5.14 also shows no sign of RTD recovering from the initial errors. It keeps moving in the wrong direction within the given *pathlimit*. RTD uses both LSS-LRTA and D*-Lite, so the selection of the wrong actions to solve $P2$ is mainly due to backward search (or D*-Lite). The main reason for the failure of the backward search, in the case of $P2$, is the large unseen and impassable area between the current location of the planning agent and the goal location. RTD performs poorly if the obstacle ratio of the map is large.

5.1.3.3   Orz103d

The best results of the five planners in the static world of Orz103d map are shown in Table 5.5.   These results show that the MOCART-CAS planner is the best planner for the Orz103d map with respect to time per search. MOCART-CAS selects a faster action selection mechanism that other planners. This highlights the importance of the use of corridors in the Monte-Carlo policy rollouts. The Orz103 map has narrow and wide streets which connect the regions. MOCART and its variations perform
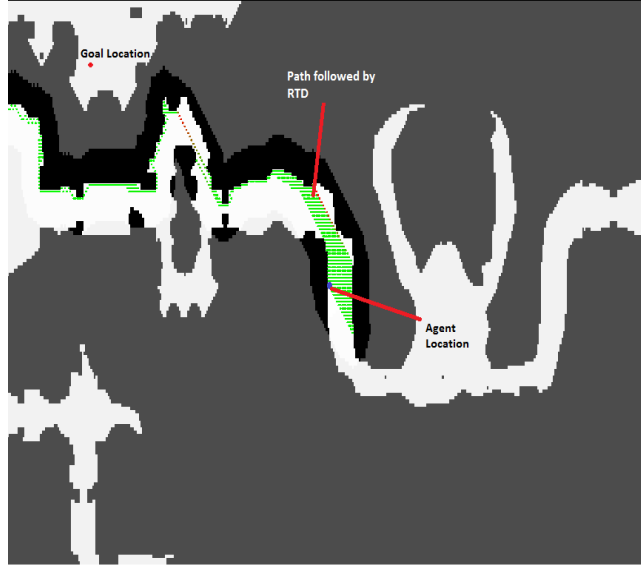
Figure 5.14: Path followed by RTD to solve $P2$.

better in finding a path via streets. This can be seen by comparing the results of MOCART and its variation on Arena2 (given in Table 5.3) and Orz301d (in Table 5.5). If the area of a local maximum in a planning problem is of a small size, then the Monte-Carlo planners can escape it quickly. To discuss it in detail, we select two planning problems as an example to explain this observation. The first planning problem, $P3$, is shown in Figure 5.15. This planning problem is an easy one that involves crossing two small narrow passages to reach the goal location and there is no local minima near the start location, so, the planners select the right action at the start of the planning. The optimal path length for this problem is 70.31. MOCART solves $P3$ with a path of length 80.15 while MOCART-RAS solves the same problem with a path of length 101.95. MOCART finds a shorter path than MOCART-RAS for problem $P3$. Figure 5.16 shows the path found by MOCART to solve $P3$ and Figure 5.17 shows the solution found by MOCART-RAS. It is obvious from these two figures
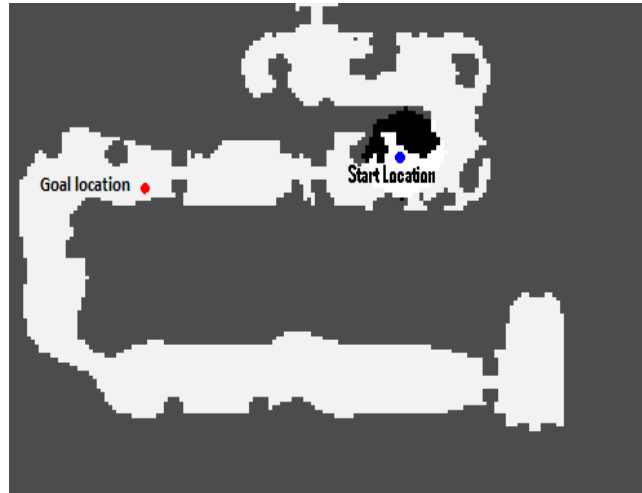
Figure 5.15: The initial and the goal locations of $P3$ on the Orz103d map.
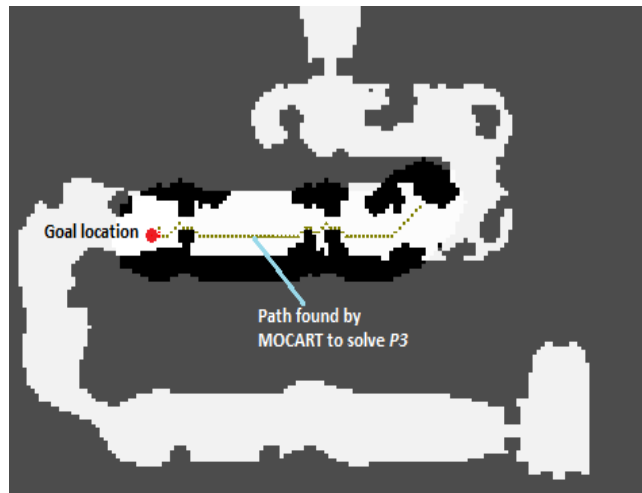
Figure 5.16: The path found by MOCART to solve $P3$.

Planner	$Depth$	$timelimit$	n_c	Avg T_s	Avg Sub	Scale
MOCART	15	30	15	0.0053	17.14	295/300
MOCART-RAS	15	30	500	0.0046	14.78	284/300
MOCART-CAS	3	30	6	**0.0004**	11.76	298/300
RTD	15	-	-	0.083	5.55	300/300
LSS-LRTA	15	-	-	0.002	3.02	300/300

Table 5.5: Time per search, Sub-optimality and Scale of five planners on Orz103d.
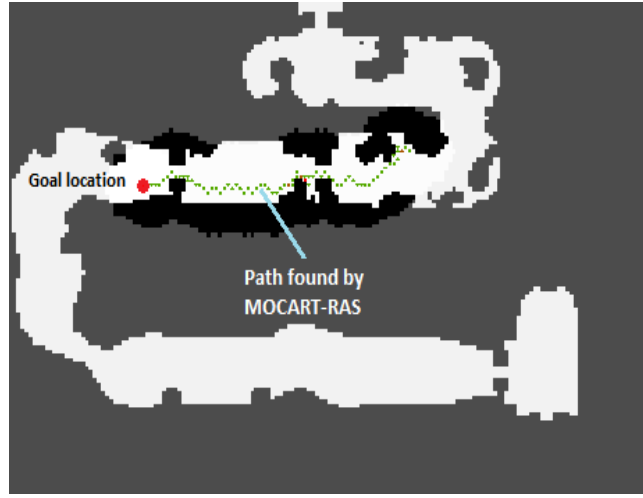
Figure 5.17: The path found by MOCART-RAS to solve $P3$.

that the path followed by MOCART is smooth and directed towards the goal while MOCART-RAS's path is in a zigzag pattern. The zigzag pattern of the path found by MOCART-RAS is mainly due to the random action sampling in the look-ahead search. The zigzag movement of the planning agent increases the path length and also makes it unrealistic. This is the key weakness of the random action sampling as compared to the focussed sampling schemes of the MOCART and MOCART-CAS planners. The look-ahead search technique (i.e. MCRT) in the MOCART planner

142

performs rollouts that are focussed towards the goal location or towards unexplored area near the goal location. The example highlights the importance of the focussed rollouts in the Monte-Carlo simulations. The MOCART-CAS planner solves the $P3$
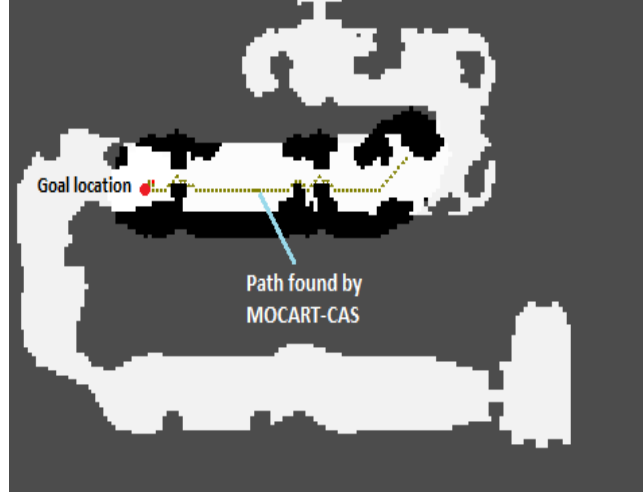
Figure 5.18: The path found by MOCART-CAS to solve $P3$.

problem with a path of length 81. The path found by MOCART-CAS to solve this planning problem is shown in Figure 5.18. The results of $P3$ show that MOCART and MOCART-CAS produce solutions of the same quality if there is no local maximum between the start and the goal locations in a planning problem. The second planning problem, $P4$, is shown in Figure 5.19. The start location and the goal locations are in two different streets. There are two local maxima near the start location on the left side of the street. There is no way to the goal location from the left corner of that street. The local maxima are highlighted in Figure 5.19. The second local maximum is a small room-like region that contains the location of the start street which has the shortest aerial distance to the goal location. The location of the first local maximum
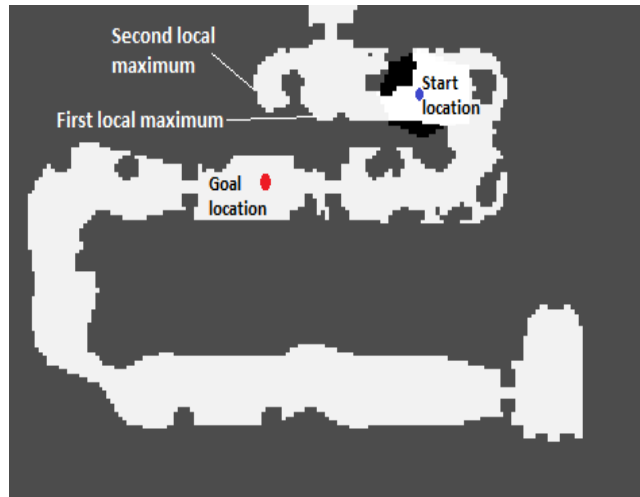
Figure 5.19: The initial and the goal locations of $P4$.

also has a shorter aerial distance than the locations on the right side of the street. Therefore, local search with an admissible heuristic at the start location guides the planning agent towards the first local maximum. The solution found by MOCART to solve $P4$ is shown in Figure 5.20. The sub-optimality of this solution is 57.54 while
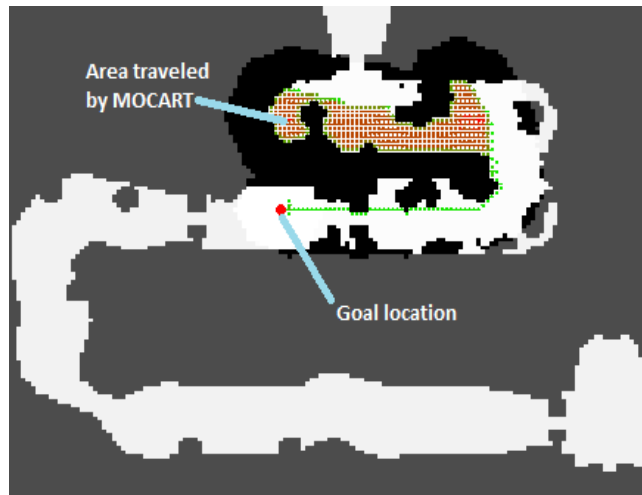
Figure 5.20: The path found by MOCART to solve $P4$.

the MOCART-RAS planner solves the same problem with a sub-optimality of 48.87. Figure 5.21 shows the path found by MOCART-RAS to solve $P4$. MOCART-CAS solves the same problem with a sub-optimality of 36.36. MOCART and MOCART-
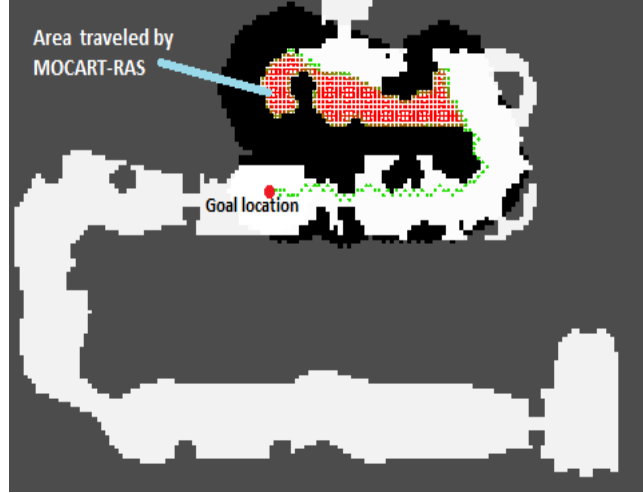
Figure 5.21: The path found by MOCART-RAS to solve $P4$.

RAS move in both local minima before they start moving towards to the goal location in the right direction. The MOCART planner takes longer than MOCART-RAS to escape the local maxima. The random sampling of MOCART-RAS also explores the actions which might not seem good according to local information, but are helpful to the global solution.

Figure 5.22 shows the path followed by the MOCART-CAS planner to solve $P4$. The figure shows that MOCART-CAS avoids getting stuck in the second local maximum; that is the reason behind the smaller sub-optimality of MOCART-CAS. The ability of the MOCART-CAS planner to avoid the second local maximum is mainly due to the pruning of the action space in the look-ahead search in a rollout. The MOCART-

CAS planner avoids exploring the corridors of the actions - which lead to the second local maximum - in the look-ahead search.
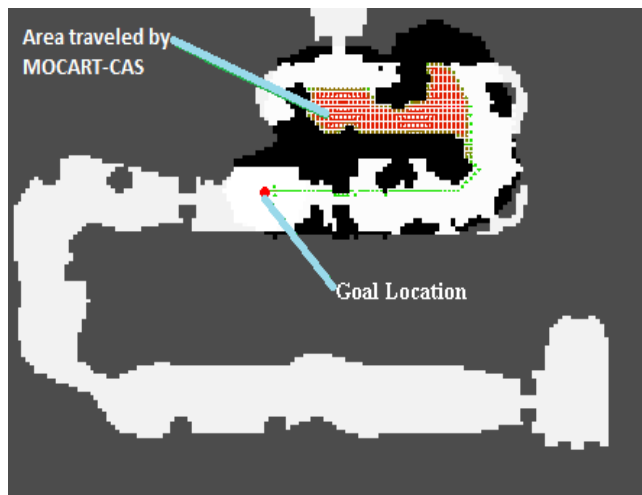
Figure 5.22: The path found by MOCART-CAS to solve $P4$.

The MOCART planner and its variations escape the local maxima within the given *pathlimit*. These local maxima are of smaller size as compared to the local maxima given in problem $P2$ (Figure 5.10). If we compare the solutions of MOCART and MOCART-CAS for the planning problems $P2$ and $P4$, it can be generalised that the quality of the solution by MOCART and its variation (for a given planning problem) mainly depends on the size of the local maximum if it exists in the part of the search space between the start and the goal locations. The action pruning in the look-ahead search helps the Monte-Carlo planner to avoid (or escape) a local maximum.

5.1.3.4   Orz702d

In the static world of Orz702d, MOCART and its variations solve all planning problems within the given *pathlimit*. A summary of average results for all planners in the

static world of Orz702d is given in Table 5.6. The results show that MOCART-CAS is significantly better than all other planners with respect to fast action selection (represented by T_s). MOCART-RAS is better than the original MOCART in time per search. RTD is a more expensive approach for the Orz702d map with respect to time per search and T_p than the other planners. However, RTD is slightly better than MOCART and MOCART-RAS with respect to sub-optimality. RTD has higher time per search and lower sub-optimality than MOCART and MOCART-RAS but it takes a considerably longer time to search for the solution. Total time to search for the solution can be higher even with the low sub-optimality because a planner sometimes does not select any action at the current state to execute in a state. This is very common in RTD. MOCART-CAS is better than MOCART and MOCART-RAS in all performance measurement parameters. This highlights the significance of the corridor sampling in the simulation model. MOCART-CAS has a significantly smaller T_s than its rival planners because of its focussed rollouts and selective action sampling.

Algorithm	$Depth$	$timelimit$	n_c	Avg T_p	Avg T_s	Avg Sub
MOCART	6	30	1500	15181.65	0.0152	2.19
MOCART-RAS	9	30	9	1214.80	0.0043	2.24
MOCART-CAS	3	30	1500	368.22	**0.0004**	2.09
RTD	15	-	-	2347.37	0.083	2.01
LSS-LRTA	15	-	-	210.37	0.050	1.19

Table 5.6: Time, Time per search, and Sub-optimality of five planners on Orz702d.

5.1.4 Results and Analysis - Dynamic World

In this section, we present the results of the experiments performed on the benchmark maps and the problem sets using the dynamic world configuration. The MAI tool supports dynamic world creation for any given map. Dynamic obstacles are created randomly by the tool. An obstacle is created and any other already existing obstacle may be removed. An example screen of obstacle creation on Arena2 is shown in Figure 5.7. The tiny black dots on the map are dynamically created obstacles. These
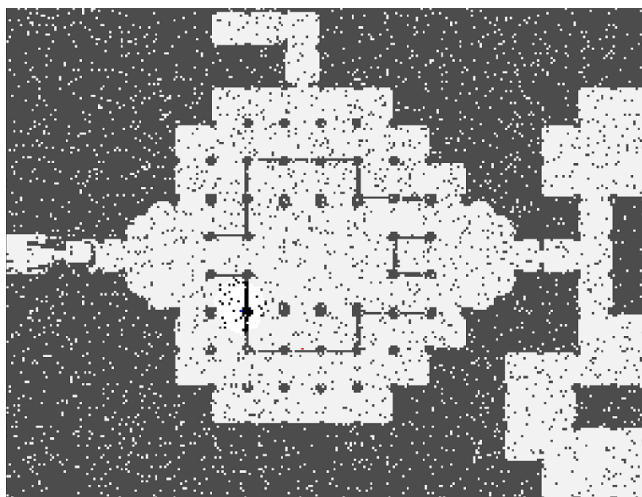
Figure 5.23: A image of the dynamic world of Arena2.

obstacles are temporary; they stay for a short time and then disappear. The tool creates (approximately 10%) new dynamic obstacles in each frame and places them randomly on the empty area and the same amount of the existing obstacles are removed to keep the obstacle ratio the same in all frames.

The results shown in the following sub sections are the average of ten runs. In all these experiments, LSS-LRTA is run with the look-ahead search of depth 15 and RTD

is run with the same depth of look-ahead search along with $localratio = 75\%$.

5.1.4.1 Arena2

In this section, we describe and discuss the performance of the planners on map Arena2 in a dynamic environment. All planners complete 300 planning problems under the given limit, i.e. $pathlimit = 100$. There is no significant difference between the performance of LSS-LRTA and MOCART-CAS with respect to sub-optimality and T_p. However, MOCART-CAS has a significantly smaller amount of time per search than all other algorithms. MOCART is the most expensive algorithm with respect to T_p and sub-optimality in a dynamic world of Arena2. If we compare the performance of RTD in the dynamic world of Arena2 with its performance in the static world of the same map, it is obvious RTD performs better in the dynamic world. LSS-LRTA's performance also improves in the dynamic world of Arena2. One of the reasons for the success of MOCART-CAS and LSS-LRTA in the dynamic world of Arena2 is the presence of dynamic obstacles that helps these two planners to escape the local minima quickly. LSS-LRTA has been designed for static worlds but it can perform well in dynamic worlds due to its capability of replanning.

5.1.4.2 Orz103d

This section provides details of the best results of the planners obtained in the dynamic world of Orz103d map. The best results of all five planners are shown in Table 5.8. The results show that MOCART-CAS is again significantly better than all other planners with respect to time per search and produces a small amount of sub-optimality. LSS-LRTA has the smallest average sub-optimality in the dynamic

149

Algorithm	*Depth*	*timelimit*	n_c	Avg T_p	Avg T_s	Avg Sub
MOCART	15	30	1500	5323.12	0.007	10.34
MOCART-RAS	15	30	500	1043.31	0.0046	3.03
MOCART-CAS	3	30	1500	155.47	**0.0004**	1.57
RTD	15	-	-	1837.14	0.013	1.69
LSS-LRTA	15	-	-	102.34	0.002	1.28

Table 5.7: Total Time, Time per search and Sub-optimality of five planners on Arena2.

Algorithm	*Depth*	*timelimit*	n_c	Avg T_p	Avg T_s	Avg Sub
MOCART	9	30	500	1404.16	0.0056	4.89
MOCART-RAS	3	30	500	480.80	0.0042	4.27
MOCART-CAS	3	30	1500	244.75	**0.0004**	2.24
RTD	15	-	-	2905.69	0.013	2.23
LSS-LRTA	15	-	-	198.23	0.002	1.42

Table 5.8: Total Time, Time per search and Sub-optimality of five planners on Orz103d.

world of Orz103d. MOCART and MOCART-RAS also find better solutions in the dynamic world of Orz103d when compared to their performance in the dynamic world of Arena2. The improvement in the performance of the Monte-Carlo planners in the dynamic world of Orz103d is due to the change in the topology of the map during problem solving. The creation of a dynamic obstacle in the local minima helps the Monte-Carlo planners to escape it quickly.

Algorithm	$Depth$	$timelimit$	n_c	Avg T_p	Avg T_s	Avg Sub
MOCART	15	30	500	3481.35	0.0073	4.34
MOCART-RAS	3	30	500	545.89	0.0043	3.96
MOCART-CAS	3	30	1500	292.61	**0.0004**	1.79
RTD	15	-	-	2565.54	0.013	1.85
LSS-LRTA	15	-	-	289.78	0.002	1.32

Table 5.9: Total Time, Time per search and Sub-optimality of five planners on Orz900d.

5.1.4.3 Orz900d

The best performance of the five planners in the dynamic world of Orz900d are shown in Table 5.9. LSS-LRTA solves the planning problems in a shorter time than the other planners. RTD and MOCART-CAS produce solutions of similar quality but MOCART-RAS solves them in significantly shorter time than RTD. LSS-LRTA is slightly better than RTD and MOCART-CAS with respect to sub-optimality. However, MOCART-CAS is significantly better than all other planners in terms of time per planning episode and produces solutions of small error (or sub-optimality). The planners perform better in the dynamic world of Orz900d than the dynamic world of Orz103d. In Orz103d, some of the streets have a very narrow entrance (or exit) and a dynamic change can keep a street blocked for some time. This kind of dynamic change increases the sub-optimality of the planner.

Algorithm	$Depth$	$timelimit$	n_c	Avg T_p	Avg T_s	Avg Sub
MOCART	3	30	500	718.56	0.0056	2.64
MOCART-RAS	3	30	500	390.34	0.0043	1.92
MOCART-CAS	3	30	1500	361.49	**0.0004**	1.59
RTD	15	-	-	1254.67	0.013	1.23
LSS-LRTA	15	-	-	174.86	0.002	1.15

Table 5.10: Total Time, Time per search an Sub-optimality of five planners on Orz702d.

5.1.4.4 Orz702d

In this section, the performance of the five planners in the dynamic world of Orz702d is given in Table 5.10. MOCART-CAS also maintains its success over other planners in terms of time per search and finds solutions of small sub-optimality. MOCART-CAS, LSS-LRTA and RTD produce solutions of almost the same quality. RTD is the most expensive planner with respect to total time to search in the dynamic world of Orz702d. MOCART and MOCART-RAS perform better in the dynamic world of Orz702d than on all other maps. These planners also perform the same in the static world of Orz702d (as shown in Table 5.6). The main reason for the better performance of MOCART and its variations in the static and dynamic world of Orz702d is the absence of local maxima of large area.

5.1.4.5 Analysis of Results

All planners perform better in the dynamic worlds of the benchmark maps as compared to their performance in the static worlds. The average sub-optimality of all

planners in the dynamic worlds of the benchmarks is shown in Figure 5.24. The results show that the MOCART-CAS planner significantly performs better than all other planners. There is no significant difference between LSS-LRTA and RTD and MOCART-CAS in terms of sub-optimality. In general, the MOCART-CAS planner outperforms its rivals. All the planners perform better in the dynamic worlds of the benchmarks as compared to their performance in the static worlds. There are two reasons behind the better performance of the algorithms in the dynamic worlds of these benchmarks.
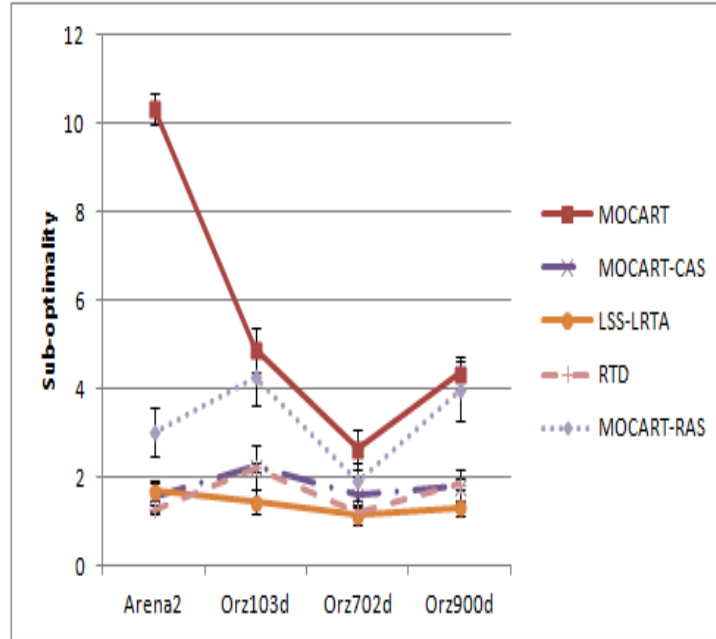
Figure 5.24: Sub-optimality of the planners in the dynamic and partially observable world of the benchmarks.

(1) The planning algorithms use the local search based on incomplete information. The estimated action values are inaccurate according to the initial static environment.

The dynamic environment can help the planner to rectify these errors by creating dynamic obstacles in the direction of an action that leads towards the local maximum (or minimum). We use a sketch of a problem ($P1$) on the Arena2 map (as an example) to explain this reason in more detail. The sketch of the problem is shown in Figure 5.25. At the start location, an optimal action, with respect to the current
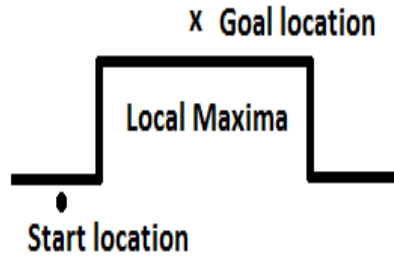
Figure 5.25: A sketch of a problem on the Arena2 map with local maxima.

neighbourhood, will always move the planner towards the local maxima. Suppose the planning agent selects such an optimal action and moves a few steps towards the local maxima. As the environment is dynamic, at the new location, the planning agent finds a set of obstacles which block its way to the local maxima (as shown in Figure 5.26). In this kind of scenario it is suitable for all planners including LSS-LRTA to
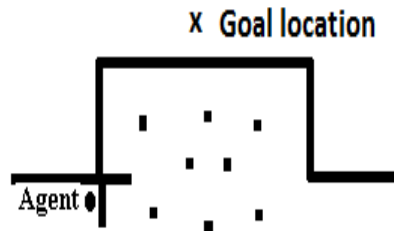
Figure 5.26: A sketch of the Arena2 map with new dynamic changes.

avoid moving into the local maxima (or minima) and find a globally optimal path.
(2) The second reason for the better performance of all planners in dynamic environment is the change of the topology of a map (during problem solving) by the tool in such a way that the new change can eliminate a local maximum from a problem. The tool removes some of the obstacles that were initially part of the map and offers a challenging scenario for the local planners. For example, in the sketched problem (shown in Figure 5.25), the wall creates the local maxima for the given problem. Suppose, the planning agent moves into the local maxima. The tool, in some cases, eliminates some parts of the wall, such that the planner can easily move to the goal location. Figure 5.27 shows a dynamic change that can help the planning agent to escape the local maxima easily. The AI research tool, called Open Real-Time Strategy (ORTS)

Figure 5.27: A dynamic change in the Arena2 map.

game engine (discussed in the next section), does not remove the existing static obstacles during problem solving and makes a better choice for the exploration of the AI planning algorithm to solve problem with partially observability and dynamicity.

5.2 The MG-MOCART Planner and ORTS Game Experiments

ORTS is an open source RTS game engine that is available for research purposes [19]. It is suitable for exploring AI for real-time strategy games. This tool is built on a client/server architecture where the game definition is kept by the server and different AI clients can be connected to the game server. To design a RTS game in ORTS, the tool requires a script file that contains information about the number of players and the positions of the movable characters (e.g workers, soldiers, sheep, bugs and tanks etc) and the structures (e.g. buildings, minerals and control centre etc) of each player (in the game) on grid. This script file is defined on the server side of the tool. The server compiles and runs the game script. Each player can see only a small part of the map in a game. A client selects the actions for one or more of its characters at the current state of the game world and sends it to the server. The server then executes these actions. This process is done in a loop called the game loop. Each iteration of the game loop is called a frame.

ORTS has a simple kinematics model. The movement of a character depends on its speed and the heading (or direction). The movement of a character is kept clocked, i.e., the movements are performed in time intervals. In each time interval (also called a frame), the character moves to another location depending on its current speed and direction (or heading). If two moving objects are about to collide in a time interval then the server stops them.

We designed a simple resource collection RTS game in ORTS to explore MG-MOCART planner in a typical RTS game. We call this game Resource Collection-RTS (or RC-

RTS in short). The details of the game are given in the next section.

5.2.1 RC-RTS Game

RC-RTS game is resource collection RTS game. It is characterised by multiple goals, partial visibility, and a non-deterministic action model. It has dynamic objects (e.g. tanks and bugs, which move randomly), and static obstacles (i.e. ridges, water ponds, nurseries, geysers, communication towers and military barracks). The game has three workers which are controlled by an AI planner. The AI planner (also called the AI player) finds a path for each worker to move to a goal location (e.g. a mineral mine). The worker picks up ten minerals when it reaches the mine. The planner then moves that worker to a control centre where the worker stores the minerals. After storing the minerals, the worker again moves to the mineral mine. This process continues until the game time is over. With their vision restricted to only eleven tiles in any direction, the workers must find a path from their start location to the mineral mine or from the mine to the control centre. If a worker reaches the mineral mine, the AI player gains ten points. The AI player earns twenty points if a worker successfully returns minerals to the control centre. The game is run on three different hand crafted maps. A view of a part of a game map is shown in Figure 5.28. The dynamic obstacles keep moving randomly around the map and frequently appear in front of a moving worker. A worker stops moving if there is a dynamic obstacle in front of it. The appearance of a dynamic obstacle in front of a moving worker makes the actions non-deterministic. This sort of activity (i.e.resource collection) is a common feature in commercial RTS games (e.g. *StarCraft II: Heart of the Swarm* by Blizzard Entertainment).
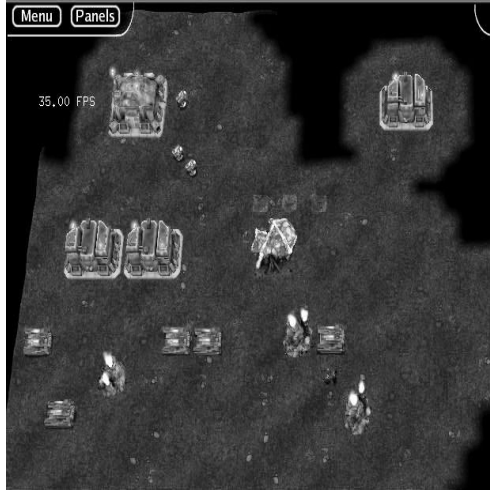
Figure 5.28: A 3D view of RC-RTS with map 2.

5.2.2 Experimental Details

We have designed a set of experiments which test the MG-MOCART technique against four of its main rivals: RRT, UCT, LSS-LRTA and RTDP. Ten tests have been performed on three different game maps; each with a grid of size 60×60. The criteria that we have used to evaluate the performance of each planning method, within the RTS environment are: **Score** – which measures the total amount of mineral recovered by the workers and **Planning Cost** – which represents the total number of states visited by the planner during the planning process for the whole game. As success in many games is measured by who gets the highest score, we naturally consider the first of these to be the most important performance indicator. The level of difficulty is controlled by constructing maps with differing numbers of static and dynamic obstacles. According to [49], a map (of any size) with an obstacle ratio between 35% and 43% is the most complicated for a planner to solve a path planning problem. The complexity of the environments created for each of the three test maps used in

158

Map	Static Obstacles	Dynamic Obstacles	Ridges	Water Tiles	Obstacle Ratio
Map 1	12	9	5	3	29%
Map 2	16	10	5	3	32%
Map 3	17	16	5	3	40%

Table 5.11: Environment variables set for each test map.

our experiments are shown in Table 5.11. Map 3 is more challenging than the other maps due to its higher obstacle ratio.

5.2.2.1 Parameter Selection

Each of the planners (except RRT) require some parameters to be tuned off-line for an application domain, and in particular we found that different planners perform better for different look-ahead depths. We selected the best possible values of the parameters for each algorithm. Several experiments have been run to find the best value for the trade-off parameter C_p of UCT. UCT performs best with $C_p = 0.1$. A look-ahead depth of four has been found to be the best for MG-MOCART and UCT. RTDP performs best with a look-ahead search of depth seven. A look-ahead search of depth nine has been found the best for LSS-LRTA. The MG-MOCART planner performs the best with weight factor $W_d = 6$. W_d is used in the reward function of MCRT (Figure 3.5, Chapter 3).

5.2.2.2 Implementation

All of the planners in this set of experiments are implemented in C++. For a fair comparison, the rival planners have been slightly modified. These modifications are made to make these algorithms applicable in the RC-RTS game. The details of the modifications of these planners are as follows.

RRT has been implemented as detailed in [44]. The RRT structure is expanded heuristically, using random samples, towards a nearest neighbour. Euclidean distance is used as the shortest distance heuristic. This means that given the current state, a neigbouring state that is nearest to the random state is selected and added into the tree if it is collision free (i.e., not occupied by a static obstacle). RRT is modified to re-use the constructed tree to plan the path to a goal if the goal state is on the RRT tree (similar to the MG-MOCART planner).

UCT [39] has been implemented with two modifications. The first modification is a limit on the look-ahead search in the policy rollout (i.e. the depth of the look-ahead search is kept fixed). The reason for this modification is to make UCT a real-time planning algorithm which can work under tight constraints (like MG-MOCART planner). The leaf-node of the look-ahead search is evaluated using the same reward function as given in MG-MOCART. This variation of UCT is also similar to the work presented in [3] for a RTS game. The similarity is in the use of a reward function to evaluate the leaf nodes.

The second modification of UCT is made to re-use the outcomes of previous searches made by UCT to reach a goal state. This change is similar to the use of tree in the RRT and the MG-MOCART planners. The estimated action values for each goal are

stored in a separate hash table for future use.

RTDP is implemented using the details given in [5] and [13]. The state transition probabilities are updated online as the environment is partially observable. In our implementation, the policy values are updated using a fixed number of iterations to guarantee a response within the strict time constraint. Euclidean distance is used as the initial heuristic in all RTDP simulations. RTDP maintains a separate hash table for each goal to store the policy values and re-use these values if the same goal is seen in the future by RTDP. This change makes it similar to the UCT, RRT and MG-MOCART planner with respect to re-use of previously learned information.

To implement LSS-LRTA in RC-RTS, we modify the A* implementation given with the ORTS download. The ORTS tool provides an efficient priority queue implementation which is used to maintain the OPEN and the CLOSED lists. LSS-LRTA maintains these lists for each goal and these lists are re-used if this goal is seen again in future. There is no change in the algorithmic details of LSS-LRTA. The re-use of the OPEN and CLOSED lists for each goal makes the LSS-LRTA planner similar to the RTDP, UCT, RRT and MG-MOCART with respect to exploiting the already learning information.

The goal assigning task is simple. Each worker has its own goal. At the start of the game, all workers are assigned the same goal, i.e. the mineral mine. Once a worker reaches the mineral mine (and picks them up), the planner changes the goal of the worker to Control Centre. If a worker reaches Control Centre, the planner changes the goal of the worker to the mineral mine. This process of the goal assigning continues until the game is stopped.

5.2.3 Results and Analysis

A summary of the scores for each planner in the test games is given in Table 5.12. We can see that in all of the test games, the MG-MOCART technique, with the two-stage planner, achieves better observed scores than the other planners. The use

Planner	Map	Minimum score	Maximum score	Mean score
MG-MOCART	1	540	1130	752
	2	170	830	507
	3	280	850	486
UCT	1	30	150	80
	2	0	150	50
	3	0	160	52
RTDP	1	20	110	50
	2	0	70	28
	3	0	70	18
RRT	1	30	210	117
	2	30	360	179
	3	10	130	72
LSS-LRTA	1	190	290	230
	2	50	160	87
	3	20	80	57

Table 5.12: Scores for each planner on maps 1-3.

of an incrementally built tree (called RT) speeds up path planning. The size of the

tree built by the MG-MOCART planner is smaller than the tree built by the RRT planner because the MG-MOCART planner adds only the collision-free nodes into the incremental tree if they are found to be promising by the policy roll-out. This reduces the size of the tree structure by keeping only the useful nodes. The small size of the tree built by the MG-MOCART planner also minimises the time required to update it if the game world is changed during the game play. This is the reason that the MG-MOCART planner performs better than RRT.

It is notable that the minimum scores of the MG-MOCART planner are higher than the maximum scores of its rivals; the closest rival is LSS-LRTA. Furthermore, the deterministic planner LSS-LRTA performs better than RRT, UCT and RTDP. This is due to the way in which A* is used to expand the look-ahead search. In general, it is observed that the scoring performance of all the planners reduces as the difficulty level of the map increases. The planning cost of the planners for all test games are shown in Table 5.13. We note that MG-MOCART's minimum planning costs are significantly smaller than those of its rivals. This is because of the small amount of search effort required by the planner to plan a path. The MG-MOCART planner is able to score higher than its rivals, whilst keeping the planning cost low, because of the way in which it uniquely re-uses the outcomes of the previous searching effort. This is achieved through the use of the incrementally built tree (of collision free nodes) which is later re-used by the MG-MOCART planner to achieve subsequent goals. RTDP is also shown to have a small planning cost when compared to its rivals. However, we do not see a correspondingly high score as we do with MG-MOCART. The average planning cost of MG-MOCART is lower than both RRT and LSS-LRTA. The RRT planner produces the highest planning cost due to its sampling approach,

163

Planner	Map	Minimum Cost	Maximum Cost	Mean Cost
MG-MOCART	1	320	1701	994
	2	225	1766	1251
	3	288	1789	1271
UCT	1	1064	2017	1655
	2	490	1765	1287
	3	496	1801	1348
RTDP	1	485	629	542
	2	439	619	520
	3	505	656	568
RRT	1	1257	2014	1885
	2	1390	2005	1812
	3	1261	2020	1884
LSS-LRTA	1	966	1893	1402
	2	1512	2191	1798
	3	1403	2208	1637

Table 5.13: Planning Cost for each planner on maps 1-3.

i.e., exploring the state space based on random samples. The results also show that the planning cost of MG-MOCART is related to the difficulty level of the map; the higher the difficulty levels the higher the planning cost. UCT keeps a balance between exploration and exploitation of actions during the simulations, therefore, its planning cost is smaller than RRT and LSS-LRTA. However, we observe from Table 5.13 that the minimum planning cost of UCT has dropped for maps 2 and 3. This is due to

the fact that when the complexity of the maps increase (i.e., an increase in static and dynamic obstacles) the planner is more likely to become stuck in local maxima (or minima). This is further evidenced by the scores shown in Table 5.12, where the zero scores indicate no goals achieved. In general, the performance of all planners degrade on maps 2 and 3. This is due to high obstacle ratio of the maps 2 and 3.

The key to the success of the MG-MOCART planner is the re-use of RT after the first goal is achieved by the planner. Figure 5.29 shows the profile of the scores achieved by the MG-MOCART planner (in each frame) on map 3. The figure shows that the score of the MG-MOCART planner increases persistently after 900 frames (or when the first goal has been achieved). The persistent increase in the scores is due to the re-use of the outcomes of the previous searching efforts. The outcomes are stored in the form of a tree (i.e., RT). in the second stage, the planner traverses RT to reach a goal, therefore, it has less search to do in a planning episode. The increase in the scores within a small number of frames also show that the planner requires less number of planning episodes to reach a goal in the second stage (because there is no cycle in RT).

The profiles of the score of other planners, on map 3, are shown in Figure 5.30. According to Figure 5.30, three planners (LSS-LRTA, UCT and RTDP) reach the first goal earlier than the MG-MOCART planner but these planners take longer to reach the other goals. LSS-LRTA has a sharp increase in the scores at the start of the game but gradually the performance of LSS-LRTA degrades with the passage of time. This is due to collisions with dynamic obstacles. LSS-LRTA monotonically increases the heuristic values of the states in the neighborhood of the current state of the planning agent. Due to collisions with dynamic obstacles, LSS-LRTA updates the heuristic
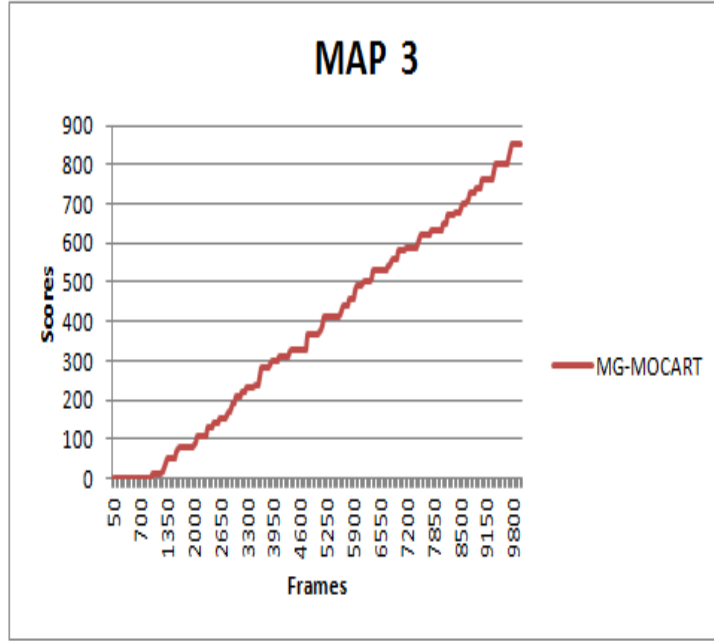
Figure 5.29: MG-MOCART planner: Score Vs Frames.

value of the same state in two or more consecutive planning episodes. For example, if a tank stays in front of a worker for a few frames, then the worker (controlled by LSS-LRTA) cannot move. In this scenario, LSS-LRTA will keep modifying (in each frame) the heuristic value of the locations surrounding the worker (planning agent). This will create a heuristic depression in the region around the planning agent. As the planner re-uses the previously learned heuristic values, it will avoid moving through this region in future (due to the high heuristic values of the states in this region). LSS-LRTA moves in cycles in the presence of a heuristic depression, therefore, it takes longer (i.e. large number of frames) to reach a goal. In a dynamic environment (such as the RC-RTS game), the performance of LSS-LRTA degrades with the passage of time due to the unnecessary modifications of the heuristic values.

The higher planning cost of the RRT planner shows that the tree generated by the
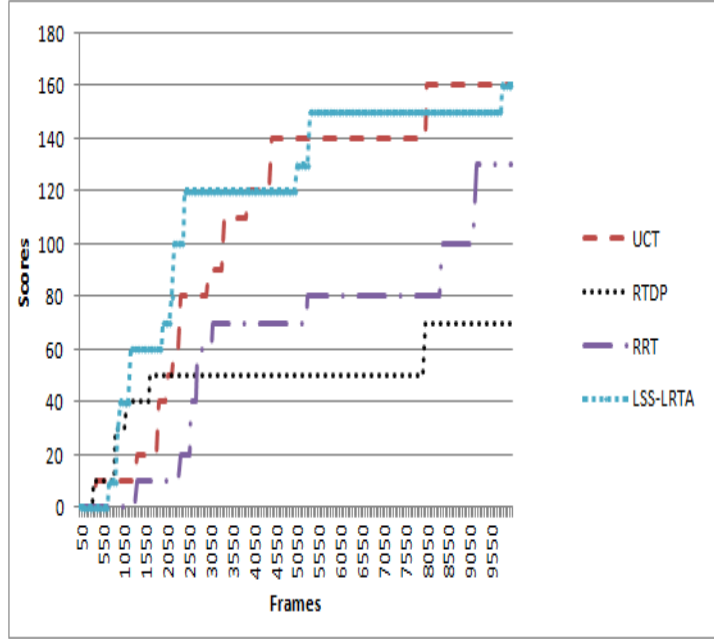
Figure 5.30: Rival planners: Score Vs Frames.

RRT planner is considerably larger than RT. This is also the reason for the lower scores of the RRT planner than the MG-MOCART planner.

The minimum searching efforts by RTDP can be explained by the fact that the RTDP planner explores only a limited part of the state space during the look-ahead search. RTDP exploration is based on the greedy action selection approach, therefore, it explores the states which look promising. Due to the greedy action selection of RTDP, it can achieve the first goal faster than other planners (as shown in Figure 5.30). The performance of RTDP degrades (with respect to score). Due to the presence of dynamic obstacles and its greedy action selection (in the look-ahead search), RTDP mostly keeps moving in cycles for a long time and achieves lower scores than the other planners.

5.3 Discussion

The results in the static world of the benchmark maps show that the MOCART-CAS planner performs significantly better than the other planners in terms of time per search. It finds solutions of similar quality to RTD, a recent real-time heuristic and incremental search path planner, on two maps out of four. It has fewer chances of getting trapped in local minima than MOCART and MOCART-RAS. MOCART-CAS is superior to its predecessors i.e. MOCART and MOCART-RAS on all maps. LSS-LRTA finds solutions of the best quality on three maps while it suffers from the problem of scalability on the Arena2 map in a static world setting where RTD can solve all problems with a very small amount of sub-optimality. One of the reasons for the success of RTD in the world of Arena2 is the presence of thin walls of static obstacles where the backward search can easily see the states around the goal state. In general, RTD performs the best if the topology of the map contains room-like regions and LSS-LRTA performs the worst in this kind of topology. If the space between the start and goal locations is highly populated by static obstacles then RTD does not perform well, this is the reason why RTD has higher sup-optimality than LSS-LRTA in the static worlds of Orz900d and Orz103d.

Another good feature of MOCART-CAS - which is consistent in all experimental outcomes - is that it performs the best with a very small time limit (*timelimit*) and look-ahead depth per planning episode when compared to its predecessors and the rival planners. MOCART-CAS requires a significantly smaller amount of planning time (T_s) per episode, than its rival techniques, to produce solutions which are comparable to RTD and LSS-LRTA.

The main weakness of the Monte-Carlo planners is the use of the raw heuristic in the reward function. MOCART and its variations do not update the heuristic (between the current state and the goal). This heuristic value is used to compute the reward function of each state transition seen in the look-ahead search. LSS-LRTA and RTD update the heuristic values and also g values (i.e. cost of moving from start location to the current location) during online planning. This is the reason for the quick recovery of LSS-LRTA from the local minima in many cases. As a future work, it is intuitive to extend the Monte-Carlo planning technique to systematically update the heuristic value of each state seen in a Monte-Carlo simulation.

The good performance of MG-MOCART planner in RC-RTS game is mainly due to the re-use of a tree of the states that the planning agent visits before reaching the goal state. In the dynamic environment of RC-RTS game, the main reason for the poor performance of LSS-LRTA is the unnecessary modification of the heuristic values due to the collisions with the dynamic obstacles. The re-use of these heuristic values can degrade the performance of LSS-LRTA instead of improving it when the planner is run for the long time in a dynamic environment.

5.4 Summary

In this chapter, two kinds of experimental setup have been presented along with an analysis of the results obtained from these experiments. In the first experimental work, MOCART and its variations were explored in the static and dynamic worlds of four benchmark domains: Arena2, Orz103d, Orz702d and Orz900d. These domains are different from each other in terms of size of the state space, the number of static

obstacles and difficulty of the planning problems. A large set of planning problems was selected from each domain for the experimental evaluation of MOCART and its variations. The empirical outcomes were compared against two recent rival path planners: RTD and LSS-LRTA. RTD has been specifically designed for dynamic worlds. The results in the static world of the benchmarks showed that the MOCART-CAS planner performed the quickest search for action selection at the current state and it produced solutions comparable to RTD and LSS-LRTA. MOCART and its variations mostly took more effort to escape the local minima than LSS-LRTA and RTD. However, MOCART-CAS was better than RTD on two maps with respect to the sub-optimality and time to search for the solutions on the static maps.

In dynamic worlds of the benchmarks, MOCART-CAS performed significantly better than all other planners with respect to the time per search and produced solutions of small sub-optimality, often close to the sub-optimality of LSS-LRTA and RTD. LSS-LRTA performed better than all other planners in terms of total time to search (T_p) in these experiments. MOCART and MOCART-RAS performed better in dynamic worlds of the benchmarks as compared to their performance in static worlds of the same benchmarks.

In the second set of experiments, the MG-MOCART planner performed significantly better than its rival planners. These experiments were conducted using a resource collection RTS game. Each planning problem in these experiments had more than one goal. The key to the success of MG-MOCART planner here is the re-use of a tree to reach a goal location that has already been visited by the planner.

Chapter 6

Conclusions

The main characteristics of the path planning problem in Real-Time Strategy (RTS) games have been defined in detail in Chapter 1. The main challenging issues for Artificial Intelligence to solve this problem are: tight real-time constraints and a dynamic and incomplete information domain world. The path planner for RTS games must guarantee to find a solution within a short time limit. Suitable path planners for RTS games interleave planning and plan execution to plan under tight real-time constraints. Interleaving planning and execution helps to uncover the features of domain world which are not visible to the planner at the start of the planning process.

The main contributions of this thesis have been described in section 1.5 of Chapter 1. One of the contributions is a Monte-Carlo policy rollout technique, called MCRT, that performs selective action sampling to generate a look-ahead search tree of fixed depth. MCRT is then used in the real-time path planners that have been designed specifically to exploit the capabilities of MCRT. These real-time path planners interleave path planning and plan execution and guarantee a solution within a fixed

time-limit. This time-limit is a user-specified parameter and is independent of the size of the state space. Two variations of MCRT, MCRT-RAS and MCRT-CAS, are presented as the second contribution of the thesis. MCRT-CAS is significantly faster than the other planners in terms of time per search (which is an important factor to achieve fast action selection in a planner).

The thesis also presents real-time Monte-Carlo path planners as the third contribution. These planners interleave planning and plan execution. Three of these planners are designed to solve single journey path planning problems. These planners are MOCART, MOCART-RAS and MOCART-CAS. The MOCART planner exploits the MCRT technique while the MOCART-RAS and MOCART-CAS planners exploit MCRT-RAS and MCRT-CAS techniques respectively. The fourth planner – called MG-MOCART – is designed to solve multiple journey path planning problems. MG-MOCART uses the MCRT technique.

Popular path finding algorithms like A*, D*, IDA*, navigational-mesh, way-points and anytime A* have been discussed and reviewed (with respect to the path planning problem in RTS games) in section 2.1 of Chapter 2. These techniques are not suitable for path planning in RTS games because these algorithms do not guarantee a solution within a fixed time interval which is independent of the size of the state space. In other words, these path planning techniques are not real-time. The variation of A* that uses the abstraction of the state space (e.g. Partial Refine A*) guarantees to find a solution within a given time-limit but these techniques are not applicable for initially unknown game maps. Learning Real-Time A* (LRTA) and its variations are suitable for path planning in RTS games for holonomic motion planning. The recent variation of LRTA (called LSS-LRTA) and a real-time variation D*-Lite (called RTD)

are selected as the closest rivals for MOCART and its variations.

The details of the MCRT technique and its two variations (MCRT-RAS and MCRT-CAS) have been described in Chapter 3. The upper bound on the number of times an action is sampled at a state depends on the convergence of the action value of that action. The convergence of an action value has been described in section 3.6 of Chapter 3. If all actions of a state have been converged then the state is declared as converged. MCRT and its variations do not run simulations for the converged state and use the converged values of the actions to choose an action. The time complexity of each technique shows that MCRT and its variations are linear in the look-ahead depth.

Four planners have been designed to exploit the capabilities of MCRT and its variations. These planners are the MOCART planner, the MOCART-RAS planner, the MOCART-CAS planner and the MG-MOCART planner. The first three planners are designed to solve path planning problem with a single goal. The MG-MOCART planner has been designed to solve planning problems with multiple goals. These planners have been described in Chapter 4. The completeness and the correctness of each planner have been discussed in this chapter. The planners ensures the completeness if there is no one way action leading to a dead-end in a state space.

Three planners (MOCART, MOCART-RAS and MOCART-CAS) are empirically evaluated using four benchmarks. These benchmarks are Arena2, Orz103d, Orz702d and Orz900d. The details of these experiments have been given in section 5.1.1 of Chapter 5. These three planners are compared against LSS-LRTA and RTD using three performance parameters. These parameters are time per search, total time of search to solve all planning problems in a benchmark and sub-optimality. Small val-

ues of these parameters are better. The results demonstrate that the MOCART-CAS planner performs better than all other planners with respect to the fast action selection. It produces similar quality plans in dynamic domains to RTD and LSS-LRTA. MOCART-CAS uniformly has the lowest time to search of all the algorithms in static and dynamic environments, leading to a more responsive real-time algorithm.

The MG-MOCART planner is evaluated in a resource collection RTS game. The details of the experimental setup and the results have been given in section 4.2 of Chapter 5. The MG-MOCART planner is compared against RRT, RTDP, UCT and LSS-LRTA. Performance is measured using the game score and the number of states explored (called planning cost) to solve the planning problems. Performance is better if the score is high and the planning cost is smaller. The results show that the MG-MOCART planner achieves a higher score than all the other planners.

6.1   Limitations

MCRT and its variations have been explored for path planning in a RTS game where the state space is explicitly represented in the form of a grid. MCRT and its variations are applicable in any domain that is represented in the form of a grid or in a continuous search space containing information about the position and velocity of the planning agent.

MCRT technique is not applicable in a domain if it is implicitly represented e.g. a sliding puzzle. This is due to the random state sampling of MCRT. It is not possible to draw such samples in an implicitly defined state space. However, two variations of MCRT-RAS and MCRT-CAS are applicable for both implicit and explicit domain

174

representations.

The four planners based on MCRT and it variations have limitations with respect to the domain representation. These planners do not guarantee to reach the goal state for a path planning problem if the domain world has one-way actions. For example, to find a path between two locations on a city map where some of the streets are one-way. The planner can get stuck in a one-way street that leads to a dead-end and cannot move out of this street.

The MOCART-RAS and MOCART-CAS planners are suitable for path planning in a domain represented in a MDP language e.g. Probabilistic PDDL [48].

6.2   Future Work

There are several possible extensions for MCRT and its variations. Some of them are as follows.

In future, MCRT and its variations can be extended by updating the heuristic values of the states according to the information gathered when the planning agent moves to initially unseen areas. We can use the following example to explain it further. Suppose a state has a static obstacle (adjacent to it) in the direction of the goal state, then the heuristic value can be increased by a weight value $w_d > 1$. This extension can be applied to the reward function. Learning of the heuristic values can improve the quality of the solution but the challenging issue in this regard is the balance between learning the heuristic values and maintaining success in terms of fast action selection. The learning process can slow the action selection mechanism if learning is applied on more than one action per planning episode. A possible future extension,

to speed up both learning and the action action mechanism, can be achieved by using the number of times a state is seen.

The MG-MOCART planner can be modified to speed up path planning for the problems given in the path finding benchmarks. The tree built by the planner can be used to generate a reduced representation of the state space in an online fashion. Then this reduced state space can be used to solve planning problems if the goal state of a problem is on the tree built from the previous search efforts. The idea is to solve some of the planning problems given in a benchmark by using the MOCART planner or its variations to build a tree of collision-free states seen by the planner. If the planner sees a problem such that its goal state is a node on the tree then the planner uses a tree search to solve the planning problem. The main challenging issue in this line of investigation is the design of a new representation that is memory efficient as the memory requirement for a higher level solution can grow exponentially with the increase in size of the space.

The definition of the convergence of an action value can be modified to guarantee a near-optimal solution by MOCART and its variations.

The MOCART-RAS and MOCART-CAS planners can be extended to solve planning problems in non-deterministic domains represented in PDDL, e.g. Racetrack [14] [5].

Bibliography

[1] N. M. Amato and Y. Wu. A Randomized Roadmap Method for Path and Ma-nipulation Planning. In *In IEEE International Conference on Robotics and Au-tomation*, pages 113–120, 1996.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, May 2002.

[3] R. Balla and A. Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 40–45, 2009.

[4] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Ragha-van. A random sampling scheme for path planning. *INTERNATIONAL JOUR-NAL OF ROBOTICS RESEARCH*, 16:759–774, 1996.

[5] A. Barto, S. Bradtke, and S. Singh. Learning to act using Real-Time Dynamic Programming. *Artificial Intelligence*, 72:81–138, 1995.

[6] R. Bellman. The Theory of Dynamic Programming. *Bulletin of The American Mathematical Society*, 60(6):503 – 516, 1954.

[7] D. Bertsekas. Distributed Dynamic Programming. *IEEE Transactions on Automatic Control*, 27(3):610–616, 1982.

[8] R. Bjarnason, A. Fern, and P. Tadepalli. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *Proceedings of the 19th International Conference on Automated Planning & Scheduling*, 2009.

[9] Y. Björnsson, V. Bulitko, and N. Sturtevant. TBA*: Time-Bounded A*. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 431–436, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[10] Y. Björnsson, M. Enzenberger, R. Holte, and J. Schaeffer. Fringe Search: Beating A* at Pathfinding on Game Maps. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. 2005.

[11] D. Bond, N. Widger, W. Ruml, and X. Sun. Real-Time Search in Dynamic Worlds. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 2010.

[12] B. Bonet and H. Geffner. GPT: A Tool for Planning with Uncertainty and Partial Information. In *Proceedings IJCAI01 Workshop on Planning with Uncertainty and Incomplete Information*, pages 82–87, 2001.

[13] B. Bonet and H. Geffner. Labelled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proceedings of International Conference on Automated Planning and Scheduling*, pages 12–21, 2003.

[14] B. Bonet and H. Geffner. mGPT: A Probabilistic Planner Based on Heuristic Search. *Journal of Artificial Intelligence Research*, 24:933–944, 2005.

[15] A. Botea, M. Müller, and J. Schaeffer. Near Optimal Hierarchical Path-finding. *Journal of Game Development*, 1:7–28, 2004.

[16] V. Bulitko and G. Lee. Learning in Real-Time Search: A Unifying Framework. *Journal of Artificial Intelligence Research*, 25:119–157, 2006.

[17] V. Bulitko, N. Sturtevant, J. Lu, and T. Yau. Graph abstraction in Real-Time Heuristic Search. *Artificial Intelligence*, 30:51–100, 2007.

[18] M. Buro. ORTS: A Hack-free RTS Game Environment. In *Proceedings of the International Computers and Games Conference*, pages 280–291, 2002.

[19] M. Buro. Call for AI research in RTS games. In *Proceedings of the AAAI Workshop on AI in Games*, pages 139–141. AAAI Press, 2004.

[20] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura. Online Planning for Resource Production in Real-Time Strategy Games. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 65–72, 2007.

[21] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud. Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. In *Proceedings of the 12th International Conference on Advances in Computer Games*, pages 1–13, 2009.

[22] M. Chung, M. Buro, and J. Schaeffer. Monte Carlo Planning in RTS Games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 2005.

[23] E. Copson. On a Generalisation of Monotonic Sequences. In *Proc. Edinburgh Math. Soc.*, volume 17, pages 159–164, 1970.

[24] T. Dean and M. Wellman. *Planning and Control.* Morgan Kaufmann Publishers, 1991.

[25] E. Dijkstra. A note on two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[26] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2, 1971.

[27] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann, ISBN 1-55860-856-7, 2004.

[28] D. Hamm. Navigational Mesh Generation: An empirical approach. In S. Rabin, editor, *AI Game Programming Wisdom 4*, pages 113–114. Charles River Media, 2008.

[29] E. A. Hansen, S. Zilberstein, and V. A. Danilchenko. Anytime heuristic search: First results. Technical report, Computer Science Department, University of Massachusetts, 1997.

[30] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, 1968.

[31] C. Hernandez and J. A. Baier. Real-time heuristic search with depression avoidance. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, Barcelona, Spain, July 2011.

[32] J. C. Herz, M. R. Macedonia, and N. D. University. *Computer Games and the Military : two views*. Center for Technology and National Security Policy, National Defense University, [Fort McNair, Washington, D.C.], 2002.

[33] Y. K. Hwang and N. Ahuja. A potential field approach to path planning. *Robotics and Automation, IEEE Transactions on*, 8(1):23–32, 1992.

[34] T. Ishida and R. E. Korf. Moving Target Search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 204–210, 1992.

[35] M. Kearns, Y. Mansour, and A. Y. Ng. A Sparse Sampling Algorithm for Near-optimal Planning in Large Markov Decision Processes. In *Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2*, pages 1324–1331, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[36] M. Kearns, Y. Mansour, and A. Y. Ng. A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. *Machine Learning*, 49(2-3):193–208, November 2002.

[37] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[38] D. E. Knuth and R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[39] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, pages 282–293, 2006.

[40] S. Koenig and M. Likhachev. D* Lite. In *AAAI/IAAI*. AAAI Press, 2002.

[41] S. Koenig and X. Sun. Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents. *Journal of Autonomous Agents and Multi-Agent Systems*, 18(3):313–341, 2009.

[42] R. E. Korf. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[43] R. E. Korf. Real-Time Heuristic Search. *Artificial Intelligence*, 42:189–211, 1990.

[44] J. Kuffner and S. LaValle. RRT-Connect: An Efficient Approach to Single-Query Path Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[45] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[46] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

[47] M. Likhachev, G. Gordon, and S. Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Proceedings of the Conference on Advances in Neural Information Processing Systems 16 (NIPS-03)*. MIT Press, 2004.

[48] D. Long and M. Fox. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[49] M. Mizusawa and M. Kurihara. Hardness Measures for Gridworld Benchmarks and Performance Analysis of Real-Time Heuristic Search Algorithms. *Journal of Heuristics*, 16:23–36, February 2010.

[50] M. Naveed, A. Crampton, and D. Kitchin. Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games. In *Proceedings of PlanSIG 2010. 28th Workshop of the UK Special Interest Group on Planning and Scheduling*, pages 125–132, 2010.

[51] M. Naveed, A. Crampton, and D. Kitchin. A Monte-Carlo Policy Rollout Planner for Real-Time Strategy (RTS) Games. In *Proceedings of PlanSIG 2011. 29th Workshop of the UK Special Interest Group on Planning and Scheduling*, 2011.

[52] M. Naveed, A. Crampton, D. Kitchin, and T. McCluskey. Real-Time Path Planning using a Simulation-Based Markovian Decision Process. In *AI-2011 Thirty-first SGAI International Conference on Artificial Intelligence*, 2011.

[53] A. Newell and H. A. Simon. GPS: A Program that Simulates Human Thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*. R. Oldenbourg KG, 1963.

[54] L. Péret and F. Garcia. On-Line Search for Solving Markov Decision Processes via Heuristic Sampling. In R. L. de Mántaras and L. Saitta, editors, *The 16th European Conference on Artificial Intelligence (ECAI 2004)*, pages 530–534, 2004.

[55] S. Rabin. A* Speed Optimizations. In M. Deloura, editor, *Game Programming Gems*. Charles River Media, 2000.

[56] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Representations by Back-propagating Errors*, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[57] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[58] S. J. Russell. Execution Architectures and Compilation. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1989.

[59] F. Sailer, M. Buro, and M. Lanctot. Adversarial Planning Through Strategy Simulation. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 80–87. IEEE, 2007.

[60] M. Shimbo and T. Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146:1–41, May 2003.

[61] L.-Y. Shue and R. Zamani. An Admissible Heuristic Search Algorithm. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, ISMIS '93, pages 69–75, London, UK, 1993. Springer-Verlag.

[62] R. M. Simpson and T. L. McCluskey. Knowledge Formulation for AI Planning. In *Proceedings of the 14th International Conference on Knowledge Engineering and Knowledge Management*, 2004.

[63] S. J. J. Smith and D. S. Nau. An Analysis of Forward Pruning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (vol. 2)*, pages 1386–1391, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[64] A. Stentz. Optimal and Efficient Path Planning for Partially-Known Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3310–3317, 1994.

[65] A. Stentz. The Focussed D* Algorithm for Real-Time Replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

[66] N. Sturtevant. Memory-Efficient Abstractions for Pathfinding. In *Proceedings of the 3rd AI and Interactive Digital Entertainment International Conference*, 2007.

[67] N. Sturtevant and M. Buro. Partial Pathfinding Using Map Abstraction and Refinement. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, 2005.

[68] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[69] R. S. Sutton. Learning to predict by the methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[70] R. S. Sutton. Reinforcement Learning Architectures. In *Proceedings ISKIT 92 International Symposium on Neural Information Processing*, 1992.

[71] G. Tesauro. Temporal difference Learning of Backgammon Strategy. In *Proceedings of the Ninth International Workshop on Machine Learning*, ML92, pages 451–457, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[72] G. Tesauro and G. R. Galperin. On-line Policy Improvement using Monte-Carlo Search. In *Proceedings of the Conference by Neural Information Processing Systems Foundation (NIPS'96)*, pages 1068–1074, 1996.

[73] C. Tovey and S. Koenig. Gridworlds as Testbeds for Planning with Incomplete Information. In *Proceedings of the National Conference on Artificial Intelligence*, pages 819–824, 2000.

[74] P. Tozour. Search Space Representations. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 85–102. Charles River Media, 2004.

[75] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

[76] R. Williams and L. C. Baird. Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions. Technical report, NU-CCS-93-14, Northeastern University, College of Computer Science, Boston, MA, 1993.