Real-Time Path Planning using a Simulation-Based Markov Decision Process

M. Naveed, A. Crampton, D. Kitchin and T.L. McCluskey

Abstract This paper introduces a novel path planning technique called MCRT which is aimed at non-deterministic, partially known, real-time domains populated with dynamically moving obstacles, such as might be found in a real-time strategy (RTS) game. The technique combines an efficient form of Monte-Carlo tree search with the randomized exploration capabilities of rapidly exploring random tree (RRT) planning. The main innovation of MCRT is in incrementally building an RRT structure with a collision-sensitive reward function, and then re-using it to efficiently solve multiple, sequential goals. We have implemented the technique in MCRT-planner, a program which solves non-deterministic path planning problems in imperfect information RTS games, and evaluated it in comparison to four other state of the art techniques. Planners embedding each technique were applied to a typical RTS game and evaluated using the game score and the planning cost. The empirical evidence demonstrates the success of MCRT-planner.

1 Introduction

Real-Time Strategy (RTS) games are complex real-time concurrent systems where players build societies and engage in simulated combat to capture territory and explore the game world to collect resources. Path planning is a challenging task that is required frequently in RTS games by human or AI players. The key challenging aspects of path planning problems in RTS games are tight time constraints, limited CPU and memory, partial visibility, and large and dynamic game worlds. These domains require an automated planner to solve planning problems with non-

Munir Naveed, University of Huddersfield, e-mail: m.naveed@hud.ac.uk,
Andrew Crampton, University of Huddersfield, e-mail: a.crampton@hud.ac.uk,
Diane Kitchin, University of Huddersfield, e-mail: d.kitchin@hud.ac.uk,
Lee McCluskey, University of Huddersfield, e-mail: t.l.mccluskey@hud.ac.uk

deterministic effects. Markov Decision Process (MDP) is a common planning formalism used to represent planning problems in nondeterministic domains. MDPs can be solved using either Dynamic Programming (DP) [3] or Monte-Carlo Simulations [20]. The current dynamic programming based planners such as mGPT [6] are applicable in domains that are modeled in a specific planning language (e.g. Probabilistic planning domain language) and all transition probabilities in the domain must be available before planning is started. The transition probabilities are used in DP to calculate the state values. Monte-Carlo (MC) planning requires only a simulation model that can generate a sequence of samples (of states or actions) according to the desired sampling distribution and a reward function to evaluate them. MC Planning uses the average rewards (of the samples) to estimate action (or state) values rather than using the pre-computed transition probabilities in a stochastic domain. Therefore, MC Planning does not need the availability of transition probabilities before the start of planning. The modeling of a RTS game in a planning language is a challenging task. MC planning is suitable for RTS games as it is easy to build a simulator for the game.

The main challenging issue in MC planning is the adjustment of the balance between the exploration of new actions and exploitation of the previously discovered promising actions. A recent MC planning approach called Upper Confidence bounds applied to Trees (UCT) [14] uses the selective action sampling approach to control the trade-off between exploration and exploitation. UCT exploits the best actions to generate the look-ahead search tree in a Monte-Carlo simulation. An action is best if its estimated value is higher than other applicable actions at a state. UCT maintains exploration by using a domain-dependent constant and the number of times an action is sampled since the start of the simulations. UCT has been an effective planning approach in domains where the reward of the state-action pairs is in the range of [0,1] e.g. Go [10]. In RTS games, the action values can be greater than this range [1]. UCT in its original form is not suitable for RTS games. The variations of UCT have been explored in RTS games for tactical assault planning [1] and path planning [19]. In this paper, we extend the work of [19] with the following contributions:

1. We introduce a new domain-independent way of controlling the trade-off between exploration and exploitation in the Monte-Carlo simulations. The new approach combines the exploration feature of rapidly-exploring random trees (RRT [18]) with the exploitation scheme of UCT. The look-ahead tree in a simulation is built by performing exploitation and exploration in a sequence. The new approach is called MCRT search.
2. The paper presents a new path planner - called MCRT Planner - that uses MCRT search. MCRT planner is a real-time path planner that interleaves planning and plan execution. MCRT planner also incrementally builds a tree to reuse the searching efforts.
3. We provide an empirical study of MCRT planner in a typical RTS game and compare its performance with its close rivals: UCT, RRT, Real-time Dynamic Programming (RTDP) [2] and LSS-LRTA* [15]. The performance is evaluated using the game score and the number of states explored during the planning.

The experimental results demonstrate the success of MCRT planner over its rival techniques.

2 Problem Formulation

The class of problems we consider are typical of autonomous agent path planning in RTS games. It is assumed an agent knows the size of the world, its position and velocity (collectively called its state), the position of its target goal states, and the set of obstacles that are within the pre set limits of its sight (which takes into account occlusion by obstacles blocking places it would normally see). The set of obstacles include both static and dynamic obstacles. The dynamic obstacles make the agent's moves non-deterministic. We formulate the path planning problem as a simulation-based Markov Decision Process (MDP) [20]. It is represented as (S,A,T,Q,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, $Q(s,a,h)$ represents the estimated value of the action $a \in A(s)$ over a finite horizon h at state s, s_o is the initial state and G is the set of the goal states. 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 state s_{next} randomly from all successor states of s that are possible to reach with a.

$$s_{next} = T(s,a) \tag{1}$$

An action a is encoded as (dx,dy,u) where $dx \in \{-1,0,1\}$,dy $\in \{-1,0,1\}$ and u is the speed of the movable agent. $V(s)$ represents the value of the state s.The value of a state s is measured using the estimated action values.

$$V(s) = max_a Q(s,a,h) \tag{2}$$

The stochastic transition function T is built using a probability distribution P. P is used to estimate the transition probabilities in the state space. 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. P is updated during online planning using a frequency based approach [2] and is given in (3).

$$p(s_i,a,s_j) = N(s_i,a,s_j)/N(s_i,a) \tag{3}$$

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 selected at s_i since the start of the planning task. The exception to this is that a move to an *occupied* state has the probability 0. In stochastic transition function T, the states with high transition probabilities have more chances of selection than the states with low probabilities.

3 The MCRT Search

MCRT search has been designed to find a new domain-independent way of handling the exploration and exploitation trade-off in the Monte-Carlo simulations and to introduce a multi-objective reward function. MCRT search evaluates an action with respect to two objectives. The first objective is to reduce the distance to the goal state and other is to avoid the collision with the static objects. Tuning the domain specific parameters (e.g. UCT uses one parameter) to balance the exploration and exploitation in a domain requires an offline empirical work. Such empirical work is a cumbersome and time-consuming task in RTS games. An investigation to find a domain independent way of managing the exploration and exploitation is crucial for RTS-type domains. Path planning in RTS games is not only required to find the shortest path but also to handle other issues e.g. to avoid areas occupied by static objects or enemy units. Intuitively, a multi-objective reward function is suitable for RTS games.
MCRT is a local search algorithm that is designed for planners that interleave planning and plan execution. MCRT search starts by finding the local search space for the current state of the planning agent. The size of the local space for a state s is $|A(s)|$. The value of each neighbouring state in the local space is measured at least once using the Monte-Carlo simulations. The Monte-Carlo simulations are performed for a fixed time duration. The simulation time is independent of the size of the search space. In every simulation, the look-ahead depth is kept finite and fixed to keep MCRT a real-time search method. In the simulations, the non-deterministic transition function is used to estimate the value and effects of sampled action. The value of each state is calculated using (2). At the end of the simulations, MCRT selects the action to move the agent to the neighbouring state that gains the highest value since the start of the simulations. Ties are broken by random selection.

3.1 Algorithmic Details

The MCRT technique borrows the idea of the generic Monte Carlo algorithm given in [14] and integrates it with the RRT sampling technique [18] for applications in path planning. Based on the problem formulation, the current state (i.e. s_c), and the current target goal (g), the MCRT function (Figure 1) returns the action it evaluates to be the most promising. 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 1) by initialising the set S of $(state, reward)$ pairs, where $state$ is a neighbour of s_c, the current state. The repeat loop iterates as long as the real-time constraints allow. The *ChooseNeighbour* function (line 3) selects which s_n 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 reward value currently recorded in

S. After each call to the *RewardSim* function is made (line 4), S is updated with a new reward value for a particular neighbour s_n. After the simulations have finished, line 7 determines s_{best}, the neighbour with the maximum estimated reward. Line 8 returns the action a which is aimed in the direction of neighbour s_{best}. We assume a is unique as the actions are directional (i.e. only one action at s_c is for the motion in the direction of s_{best}). Key to MCRT is the simulation procedure *RewardSim*

Function $MCRT(s_c, g)$
Read access $depth$,$timelimit$;
1. $S := \{(s,0) : s \text{ is a neighbour of } s_c\}$;
2. REPEAT
3. $s_n := ChooseNeighbour(s_c)$;
4. $r_n := RewardSim(s_n, g, depth, 1)$;
5. Update S with (s_n, r_n)
6. UNTIL $(ElapsedTime() > timelimit)$;
7. find $(s_{best}, r_{max}) \in S : (s,r) \in S \Rightarrow r_{max} \geq r$;
8. RETURN the action a that aims towards s_{best}

End *MCRT*

Fig. 1: High Level Design of MCRT

(Figure 2) which estimates the reward of moving to s_n. This is adapted from the generic Monte Carlo algorithm given in [14]. The latter work introduced a bandit algorithm for this, whereas we adapt the technique to RTS games using a novel reward estimator which takes into account potential collisions, and a static estimate involving distance from the goal. *RewardSim* expands a look-ahead tree from the neighbouring state s_n of current state s_c, by generating random samples of states, and accumulating rewards as it searches, as explained below. The main recursive

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*

Fig. 2: RewardSim: MCRT Look-ahead Search

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. 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 Euclidean metric, and selects an action which is aimed towards the 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 inverse Euclidean distance from s to the target. The function *SimulateAction* returns

Function $SimulateAction(s_n, a, g)$
Read access MDP, W_d;
1. $s_{next} := Transition(s_n, a)$;
2. $rw := \|\{t : p(s_n, a, t)) > 0\}\| / (W_d * dist(s_{next}, g))$;
3. $return[s_{next}, rw]$
End $SimulateAction$

Fig. 3: Simulate Action

a 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 the execution of action a at s if $p(s_n, a, s_{next}) > 0$. $Transition$ (line 1 in Figure 3) assembles a list of the possible new states, and selects the next state s_{next} randomly from the list, 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 less obstacles present in that action's direction of travel (and thus less chance of future collisions). The size of the list is divided by $dist(s_{next}, g)$, the Euclidean distance 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 with minimising the Euclidean distance to goal states.

Figure 4 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 1). These neighbours are labeled as 1.1 in Figure 4 (in Figure 4, 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 1). 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 RRT 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 2). An action a is selected to expand E2 towards C9. The action a is simulated (line 4, Figure 2) and an outcome of a is estimated using the probability distribution. In the example, D3 is

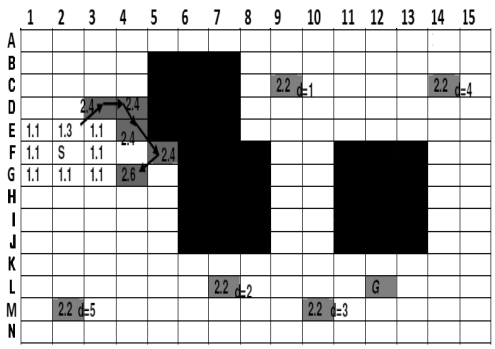

Fig. 4 An MCRT Example: *S* is the current state, *G* is the goal state and *d* is the look-ahead depth. The cells in Black are the static obstacle.

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 4, G4 is the leaf node of the look-ahead search, and is evaluated using line 6 of Figure 2. 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.

4 An MCRT-based Path Planner

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 information described in the "Problem Formulation" section above, and interleaves planning and execution as follows. Lines 1 and 2 (Figure 5) initialise a RRT tree structure (T). 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), the RRT tree structure is populated using the MCRT technique, interleaved with action execution (line 9).

The RRT structure T builds up a *tree* of the collision free states of the search space within S, as it is possible that s can be revisited if (a) states are retraced (b) $Execute(a,s)$ leaves s unchanged - where there is an obstacle. T is expanded with an edge from s to s_{next} if it passes the Valid test: this test returns true unless s_{next} already appears in T, or if s already has a child edge in T 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 the first stage achieves the first goal, the planner enters

Procedure *MCRTPlanner*
Read access MDP formulation (S,A,P,R,s_o,G);
1. initialise tree $T := null$ and state $s := s_o$;
2. $T.AddRoot(s)$;
3. WHILE G is not Empty
4. pop g from G;
5. REPEAT
6. $a := MCRT(s,g,d,n)$;
7. $s_{next} := Transition(s,a)$;
8. IF $Valid(s_{next},s,T)$ THEN $T.addnode(s,s_{next})$;
9. $s := Execute(a,s)$
10. UNTIL $s.pos = g$;
11. pop g from G;
12. REPEAT
13. $a := LocalPlanningMethod(s,g,T)$;
14. $s_{next} := Transition(s,a)$;
15. IF $ObstacleFree(s_{next})$
16. THEN $s := Execute(a,s)$
17. ELSE $T.remove(s_{next})$;
18. IF $s.pos = g$ THEN pop g from G
19. UNTIL $\neg ObstacleFree(s_{next})$
20. END WHILE
End *MCRTPlanner*

Fig. 5: MCRT-Planner

the second stage. In the second stage (lines 12 - 19), the planner exploits the fact that T has been built up in the first phase, searching T to find the path to the next goal state from the current location. The local planning method (line 13, Figure 5) 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 $Transition$ described above) changes the agent's state to a state 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.

5 An RTS game

As an application we use RC-RTS, a typical real-time strategy game that has been developed using the Open Real-Time Strategy (ORTS) game engine [8]. ORTS provides an appropriate environment for studying real-time AI problems such as path finding and imperfect information. RC-RTS is a resource (minerals) collection game characterised by multiple goals, partial observation, and non-deterministic actions.

Fig. 6 RC-RTS with Map 2.

It incorporates dynamically changing objects (tanks and bugs, which move randomly), and partially known static obstacles (ridges, water ponds, nurseries, geysers, communication towers and military barracks). An AI client controls workers who have to collect the minerals and return them to a control centre. With their vision restricted to only eleven tiles in any direction, they must find a path from their start location to the mineral cluster, pick up ten pieces of the minerals – gaining ten points for doing so – and then find a path from the mineral cluster back to the control centre. A worker who successfully returns minerals to the control centre gets a further twenty points. A sample of the game map is shown in figure 6.

6 Related Work

The path planning problem has been extensively studied in the area of computer games. Optimal path planning approaches like A* [11], way-points and navigational mesh are not applicable in the RTS games due to the time constraints and incomplete information of the game world. Learning real-time A* (LRTA)[16] is a real-time heuristic search planner that is designed for solving the planning problems in real-time. LRTA searches for an action using a look-ahead of depth one. LRTA also updates the heuristic value of the current state in a planning episode. The main drawback of LRTA is its easily getting trap into heuristic depression (HD) [13, 12]. HD makes real-time heuristic search get stuck in a small region due to the inappropriate heuristic values of the states in that region. It takes several searching efforts of the planner to escape a HD. The recent variation of LRTA e.g. LRTS [7], LSS-LRTA [15] and aLSS-LRTA [12] have been designed to escape HD. However, these variations require a lot of searching efforts to escape HD or to avoid it. LRTS increases the look-ahead depth to escape from HD. LSS-LRTA* [15] uses A* to identify the look-ahead search space of fixed depth in the current vicinity of the planning agent and then updates the heuristic values using Dijkstra's approach [9]. LSS-LRTA is faster than LRTS because it updates the heuristic values of all the edges seen during the look-ahead search. Another notable characteristic of LSS-LRTA is its better per-

formance than D* Lite on static maps. aLSS-LRTA is a variation of LSS-LRTA that avoids HD by appropriately selecting the best state (in a look-ahead search). However, LSS-LRTA and aLSS-LRTA do not decrease the action costs if required due to a dynamic change in the game world. Due to these drawbacks, LSS-LRTA and its variation can be expensive for path planning in dynamic worlds. MCRT search does not get stuck in small regions due to its exploration capabilities. The action values of a state are decreased or increased by MCRT search according to the current settings of the environment. This makes MCRT search suitable for dynamic worlds. A recent variation of LRTA called Real-Time D* (RTD*) [4] handles the problem of the increase and decrease of an edge cost due to the dynamic change in the domain. Real-Time D* uses bidirectional search, combining real-time and dynamic search, which allows it to react to dynamic changes in the world and update the heuristic values accordingly. It seems a promising approach but the main reason for not using RTD* in our domain is its dependency on backward global search. MCRT is also applicable in the high dimensional search spaces [18] due to its RRT based sampling capabilities.

7 Experimental Setup

We have designed a set of experiments which test the MCRT 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 and by introducing a successively increasing number of narrow passages and ridges. The complexity of the environments created for each of the three test maps used in our experiments are shown in table 1. Each of the planners (except RRT) require some parameters to be tuned off-line for an application

Table 1: Environment variables set for each test map.

Map	Static Obstacles	Dynamic Obstacles	Narrow Passages	Ridges	Water Tiles
Map 1	12	9	4	5	3
Map 2	16	10	5	5	3
Map 3	17	16	7	6	3

domain, and in particular we found that different planners perform better for different look-ahead depths. We decided to make the comparisons between planners with each one performing optimally. For UCT the trade-off parameter C_p was set to 0.1 (see [14]). A look-ahead depth of four was chosen for MCRT and UCT, a depth of seven for RTDP and a depth of nine for LSS-LRTA*. In our experiments, RRT has been implemented as detailed in [17]. The RRT structure is expanded heuristically, using random samples, towards a nearest neighbour. We use Euclidean distance as a heuristic to expand the tree. 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). Once the first goal is achieved then RRT re-uses the constructed tree to plan the path to subsequent goals (this is a two stage planner similar to MCRT). The resulting tree can be thought of as having a similar structure to that of traditional way-points. For UCT, we have implemented the algorithm given in [14] but with some variations. These variations are made due to the time-constraints and the presence of multiple goals in a planning problem. UCT uses the same reward function as given in MCRT. To reuse the outcomes of previous searches in UCT, the estimated action values for each goal are stored in a separate vector for future use. RTDP is implemented using the details given in [2] and [5]. RTDP formulates the path planning problem as a Markov Decision Process and tunes the policy (a mapping from states to actions) during the online search; we use hash tables to store the policy values. In our implementation, policy values are updated using a fixed number of iterations. Furthermore, we use a frequency based approach given in (3) to measure the probability distribution for RTDP, and Euclidean distance as the initial heuristic in all RTDP simulations. We use two hash tables to store the state values in RTDP - one for each goal in a planning problem - to reuse the efforts done in the previous search. To implement LSS-LRTA* in RC-RTS, we modify the A* implementation given with the ORTS download. The priority queue is implemented as a heap. 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 minerals. Once a worker reaches the mineral cluster (and picks them up), the planner changes the goal of the worker to Control Centre and sets a boolean variable as true. This boolean variable is used to decide which data structure (hash tables or vectors) is to use for path planning in the case of UCT, LSS-LRTA and RTDP. If a worker returns to the Control centre, the planner changes the goal of the worker and sets the boolean variable as false.

8 Results

A summary of the scores for each planner in the test games is given in table 2. We can see that in all of the test games, the MCRT technique, with the two-stage planner, achieves better observed scores than the other planners. The use of an incrementally built MCRT tree structure speeds up both the path planning and the motion of the workers. Although the RRT planner also incrementally builds up a similar tree dur-

ing planning in the first stage, it does not perform the same as MCRT. This is due to the huge size of the tree built by the RRT planner. The MCRT 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 the useful nodes only. The small size of the MCRT tree also minimises the time required to update it if the game world is changed during the game play. It is notable that the

Table 2: Scores for each planner on Maps 1-3.

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

minimum scores of the MCRT 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. Though LSS-LRTA* has a deterministic approach to path planning, its behaviour looks non-deterministic in our game because of the interleaving of planning and execution and because of its ability to tune the heuristic function through learning. A worker's path that is controlled by the LSS-LRTA* planner changes its direction of movement when it collides with a tank or other dynamic obstacle. In general, it is observed that the scoring performance of the planners reduces as the difficulty level of the maps increase. The planning cost of the planners for all test games are shown in table 3. We note that MCRT's minimum planning costs are significantly smaller than those of its rivals. This is a result of needing a small amount of search effort for the planner to determine a path plan. The MCRT 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 MCRT planner to achieve subsequent goals. RTDP is also shown to have a reduced planning cost when compared to its rivals. However, we do not see a correspondingly high score as we do with MCRT. 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 simulations. The reasons for the minimum exploration in this case are i) a greedy action selection approach (i.e., best action) and ii) slow convergence (of the policy values). The average planning cost of MCRT is lower than both RRT and LSS-LRTA*. The

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

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

RRT planner produces the highest planning cost due to its sampling approach, i.e., exploring the state space based on the random samples. The results also show that the planning cost of MCRT is related to the difficulty level of the map; the higher the difficulty levels the more planning cost it consumes. UCT keeps a balance between the exploration and the exploitation of the actions during the simulations, therefore, its planning cost is smaller than RRT and LSS-LRTA*. However, we observe from table 3 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 minima. This is further evidenced by the scores shown in table 2, where the zero scores indicate no goals achieved.

9 Conclusions

In this paper we have introduced a new real-time path planning algorithm which is aimed at finding paths for agents in applications as typified by real-time strategy games. Here the agents inhabit a world containing obstacles some of which continuously change positions; they have multiple sequential goal states to find, limited time to plan their next move, imperfect information about the effect of their move actions and partial information about the positions of obstacles. Our planner is founded on two key innovations:
- the MCRT search for finding the next action to execute. This technique is based on a fusion of two existing techniques (UCT and rapidly expanding random trees) together with a novel reward function which takes into account the likelihood of collisions along a path
- a two stage structure. During one stage, path finding to solve one goal using the MCRT planner builds up a RRT structure. This is then exploited in a second stage which uses a standard search technique until conditions change to invalidate the RRT, in which case the first stage is re-engaged and the tree restored.

These features of MCRT leverage the domain characteristics that multiple sequential goals have to be solved, and that reward estimates should be collision-sensitive, to make it superior to its rivals. In future work, we aim to explore the performance of MCRT planner on the pathfinding benchmark problems. MCRT is also extendable for path planning in the environments modeled as a Digital Elevation Model [21] by introducing a height parameter in the reward function.

References

[1] Balla R, Fern A (2009) UCT for Tactical Assault Planning in Real-Time Strategy Games. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence, pp 40–45

[2] Barto A, Bradtke S, Singh S (1995) Learning to act using Real-time Dynamic Programming. Artificial Intelligence 72:81–138

[3] Bellman R (1954) The theory of dynamic programming. Bulletin of The American Mathematical Society-BULL AMER MATH SOC 60(6):503 – 516

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

[5] Bonet B, Geffner H (2003) Labelled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In: Proceedings of ICAPS, pp 12–21

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

[7] Bulitko V, Lee G (2006) Learning in Real-Time Search: A unifying framework. Journal of Aritificial Intelligence Research (JAIR) 25(1):119–157

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

[9] Dijkstra E (1959) A note on two problems in connexion with graphs. Numerische Mathematik 1:269–271

[10] Gelly S, DSilver (2007) Combining Online and Offline Knowledge in UCT. In: ICML 2007, pp 273–280

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

[12] Hernández C, Baier J (2011) Real-Time Heuristic Search with Depression Avoidance. In: Proceedings of the twenty-second international joint conference on Artificial Intelligence

[13] Ishida T (1992) Moving target search with intelligence. In: Proceedings of the tenth national conference on Artificial intelligence (AAAI92)

[14] Kocsis L, Szepesvári C (2006) Bandit Based Monte-Carlo Planning. In: Proceedings of the 17th European Conference on Machine Learning, pp 282–293

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

[16] Korf RE (1990) Real-Time Heuristic Search. Artificial Intelligence 42:189–211

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

[18] LaValle S (2006) Planning Algorithms. Cambridge University Press

[19] Naveed M, Kitchin D, Crampton A (2010) Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games. In: Proceedings of PlanSIG 2010., pp 125–132

[20] Sutton R, Barto A (1998) Reinforcement Learning: An Introduction. MIT Press

[21] Wood J (1996) The Geomorphological Characterisation of Digital Elevation Models. PhD thesis, University of Leicester, UK