Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games

Munir Naveed, Diane Kitchin, Andrew Crampton

University of Huddersfield,
School of Computing & Engineering,
Huddersfield, HD1 3DH, United Kingdom
E-mail(s): {m.naveed, d.kitchin, a.crampton}@ hud.ac.uk

Abstract

In this work, we explore two Monte-Carlo planning approaches: Upper Confidence Tree (UCT) and Rapidly-exploring Random Tree (RRT). These Monte-Carlo planning approaches are applied in a real-time strategy game for solving the path finding problem. The planners are evaluated using a grid-based representation of our game world. The results show that the UCT planner solves the path planning problem with significantly less search effort than the RRT planner. The game playing performance of each planner is evaluated using the mean, maximum and minimum scores in the test games. With respect to the mean scores, the RRT planner shows better performance than the UCT planner. The RRT planner achieves more maximum scores than the UCT planner in the test games.

Introduction

Real-Time Strategy (RTS) games are the ideal platform to investigate state-of-the-art and sophisticated artificial intelligence techniques. Their suitability to AI research is because of the complex and challenging environment they offer to the players. The main challenging issues for the computer player are uncertainty, durative actions, tight time constraints and the dynamic gaming world. One of the few suitable approaches for such games is Monte-Carlo (MC) planning (Chung et al., 2005). However, there are very few studies of MC planning in RTS games. In this paper, we provide an initial investigation of two MC planning approaches for solving the path finding problem in a RTS game called RC-RTS (details of the game are given in the next section). These two approaches are UCT (Kocsis & Szepesvári, 2006) and RRT (LaValle, 2006).

The other suitable planning approaches for RTS games are the real-time heuristic search and real-time dynamic programming (RTDP). Planners based on real-time heuristic search are popular for solving the path planning problems in dynamic environments due to their capability of interleaving both planning and action execution within a fixed time interval; for example, Learning Real-Time A* (Korf, 1990), Learning Real-Time Search (Bulitko and Lee, 2006) and hierarchical task-based real-time path planning (Naveed *et al*., 2010). However, these approaches suffer from two main problems: convergence to the local minima and a slow convergence rate if the gaming environment is largely populated by static obstacles. The RTDP planner (Barto *et al*., 1991) formulates a planning problem as a Markov Decision Process and tunes the utility function (a mapping from states to actions) during the online search. RTDP planner also suffers from the problem of slow convergence speed.

The main motivation for using Monte-Carlo planning for solving the path finding problem in RC-RTS is due to the success of MC planning in Go (Lee *et al*., 2009), Solitaire (Bjarnason *et al*., 2009), sailing strategies (Kocsis & Szepesvári, 2006) and robot motion planning (LaValle, 2006). Kocsis and Szepesvári applied UCT as a non-deterministic path planner for the sailing strategies (Vanderbei, 1996) to find a path between two locations on a grid. Their results showed that UCT required fewer samples to generate near optimal path plans than the RTDP planner and a trajectory based online heuristic sampling technique (with control on the look-ahead depth) (Péret and Garcia, 2004). RRT-based path planning is particularly suitable in RTS games for the movements of characters with steering constraints, e.g. cars, tanks and airplanes. This suitability is due to the capability of RRTs to handle differential constraints.

Kocsis and Szepesvári proposed UCT as a rollout Monte-Carlo planner. In each rollout, a look-ahead tree is expanded to a certain depth with the current state always as a root node of the tree. The main contribution of Kocsis and Szepesvári is equation (1) which solves the trade-off between exploration and exploitation of the actions applicable in a state seen during the tree search. The leaf nodes of the look-ahead tree are evaluated using a random function. This search is performed for several rollouts (depending on the stopping condition) and then an applicable action (of the current state) which has the highest predicted reward is selected for execution.

$$Q_{(s,a,d)} = Q_{(s,a,d)} + C_p \sqrt{\frac{ln(N_{(s,d)})}{N_{(s,a,d)}}} \qquad (1)$$

$Q_{(s,a,d)}$ is the estimated reward of the action a at state s and depth d of the look-ahead tree. $N_{(s,d)}$ is the number of times that state s has been visited since the first rollout, $N_{(s,a,d)}$ is the number of times action a has been selected at s since the first rollout. $C_p > 0$ is a constant value which is tuned for every domain.

Rapidly-Exploring Random Tree (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 (LaValle, 2006). For example, RRT-Connect (Kuffner and LaValle, 2000) 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.

In this preliminary work, we combine the random sampling technique of RRT with the UCT's rollout Monte-Carlo planning. We do not maintain the tree structure explicitly in the memory (unlike the RRT-Connect) and only use RRT as a replacement for the UCT's selective action sampling technique. We also describe an application of UCT in our RTS game for solving the path finding problem and an empirical comparison is made between RRT and UCT planners. However, we use a heuristic based evaluation function for the UCT planner. This is because our RRT planner is using a local heuristic search method for the action selection task at a given state during the look-ahead search.

The rest of our paper is organised as follows. The next section, "RC-RTS Game", gives a brief definition of our RTS game. "Path Planning Problem" describes the formulation of the planning problem. In "UCT Planner", we provide details of the UCT planner as applied to the path planning in RC-RTS. The next section "RRT Planner" gives the description of the RRT based MC Planner. Section "Experimental Design" provides the experimental details of the research work. The results of the empirical work are given in section "Results". Finally, the conclusions along with a description of future work are given in section "Discussion".

RC-RTS Game

RC-RTS is a resource collection RTS game that we have devised and built in an open source RTS gaming platform called Open Real Time Strategy Game Engine (ORTS) (Buro, 2002). It is a single player game of imperfect information. The player (which is an AI client) has three workers and a control centre. Each worker can move to an empty place on the game map if there is no static or dynamic obstacle between both locations. RC-RTS has a single location containing a cluster of minerals. The main goal of the AI player is to collect as many of the minerals from this location as possible and to store them at the control centre within ten thousand game ticks. The player's score is increased by ten if a worker reaches the mineral cluster and picks them up. When a worker returns the minerals to the control centre then the AI player gets a further twenty points. To achieve a maximum score, the AI player is required to plan the shortest paths from the control centre to the mineral cluster and vice versa.

The game also has other movable characters which are used as the dynamic obstacles for the workers. These characters include tanks and invicor (a bug that moves in a frog like jumping style). Static obstacles are created using immovable characters and ridges. The immovable characters include nurseries, geysers, barracks and comsats. A screenshot of a run of RC-RTS with the start positions of the workers is shown in figure 1. This screenshot shows only a part of the top left side of a game map. The minerals and comsats are located near the bottom right corner of the map. Therefore they are not visible in the screenshot. The screenshot is processed to add labels with some visible characters.

Figure 1: A screenshot of RC-RTS

Path Planning Problem

Path planning for RC-RTS is a non-deterministic planning problem that can be addressed as a Markov Decision Process (MDP). We formulate this MDP as a tuple $\psi = (S, A, T, s_o, G, R)$ where S is a finite state space, A is the set of actions and T is the set of transition probabilities. The transition probability $T_a(s_i, a, s_j)$ represents the probability of moving a character from s_i to s_j if an action a is executed at s_i where $a \in A(s_i)$ and $s_i, s_j \in S$. $A(s_i)$ is a set of the applicable actions at state s_i and $A(s_i) \subseteq A$. s_o is the initial state. G is a set of goal states. R is the reward function for a state-action pair, i.e. $R: s \times a \rightarrow \mathbb{R}$.

A path plan in this case is a total function that maps a state into an action. For each planning problem at a state s, the planner runs a finite number of Monte-Carlo simulations and estimates the rewards for the applicable actions of s. The applicable action of s which gains the highest

estimated reward (from the simulations) is selected for execution. In this work, we study the UCT and RRT planners to tune the reward function.

UCT Planner

We made the following modifications to Kocsis and Szepesvári's UCT work for its application in RC-RTS for path planning.

1. The evaluation function, in our case, is the inverse of the Euclidean distance between the leaf node and a goal state (unlike the random evaluation function of UCT).
2. To simulate an outcome of an action *a* at state *s* and measure the reward for this transition during the look-ahead search, we use the function S*imulateAction* given in figure 2. A set of possible successor states S_s is populated using the function *ChildStates* (line 1 of figure 2). This function returns a list of successor states of the current state s which are reachable from s with the application of action a such that each successor state $s' \in S_s$ has $T_a(s,a,s') > 0$.

```
Function SimulateAction(State s, Action a)
    1: Vector<State> S_s=ChildStates(s, a)
    2: s' =SelectState(S_s)
    3: reward=Metric( s' , S_s)
    4: return [ s' , reward]
End Function
```

Figure 2: Action simulation

A transition probability $T_a(s,a,s')$ is always zero if s' is occupied by a static obstacle. The function *SelectState* (line 2) selects a state s' from S_s randomly as an outcome of action *a* at state *s*. The function *Metric* (line 3) is applied on s' to measure the reward of the state-action pair (i.e. $(s \times a)$). In our case, the reward is measured using equation (2).

$$reward = \frac{|S_s|}{d_{(s',G)}} \tag{2}$$

$d_{(s',G)}$ is the Euclidean distance between s' and G . The size of S_s is used to estimate the collision with the static obstacles. A small size of S_s of action *a* in the look-ahead tree means the application of this action may cause a collision in future. Therefore, the reward is kept directly proportional to the size of S_s . The relationship between the distance measure and the reward is set in an inversely proportional manner. This is because the transition of a path planning character from the current state to the next state (due to an action *a*) should reduce the distance of the character from the goal location.

RRT Planner

We apply the RRT planner as a rollout Monte-Carlo planner. The RRT planner repeatedly searches for the best neighbouring states of the current state during the Monte-Carlo search - as shown in figure 3 (line 4). Once the Monte-Carlo search is stopped, the neighbouring state with the highest predicted reward is selected as the best state s_b (line 6). An action *a* is planned (line 7) to move the character to the best neighbouring state s_b. The planned action *a* is executed (line 8) and the new state of the character is observed (line 1). If the character is at the goal state then stop path planning (line 2).

RRT planner's look-ahead search expands in a UCT style (as shown in figure 4). The look-ahead search grows up to a fixed depth *d*. If the tree reaches the leaf nodes at depth *d* then it evaluates the leaf node (line 1). The leaf nodes are evaluated using the inverse of the Euclidean distance between the leaf node and the goal state. If the look-ahead tree encounters an invalid state then the tree expansion is stopped (even before the leaf node is reached) and a zero value is assigned as an evaluation to the invalid state (line 2). The invalid states are the states which are occupied by static obstacles.

```
Function RRT Planner
    1: State s_i=GetCurrentState()
    2: If s_i=GoalState return 0
    3: While (stopping_condition)
    4:     Search(s_i , 0)
    5: End while
    6: State s_b=FindBestNeighbourState(s_i)
    7: Action a=PlanAction(s_i, s_b)
    8: Execute a
    9: Go to step 1
End Function
```

Figure 3: RRT Planner

RRT planner's tree is expanded using the valid neighbouring states of a state *s*. The S*electNeighbourState* function (line 3) calculates a set of valid neighbouring states (maximum possible neighbouring states are eight). This function selects a neighbouring state randomly if the neighbouring states are seen for the first time in the look-ahead search since the first rollout. Otherwise it returns a neighbouring state (*nstate* in figure 4*)* that has the highest predicated reward (i.e. Q_{rrt}). The function *RandomState* (line 4) selects a random valid state *rstate* from the state space S with the condition that *rstate* is not a part of the neighbourhood of the current state.

```
Function Search(state s, depth d)
    1: If Leaf(s,d) then return Evaluate(s)
    2: if(invalid(s)) then return 0
    3: nstate=SelectNeighbourState(s)
    4: rstate=RandomState(nstate)
    5: Action a=PlanAction(nstate, rstate)
    6: [nextstate, reward]=SimulateAction(nstate, a)
    7: nstate.Q_rrt=reward+Search(nextstate, d+1)
    8: return nstate.Q_rrt
End Function
```

Figure 4: RRT Search

The function *PlanAction* (line 5) is in fact a local planning method (LPM) which uses a heuristic search in the immediate neighbourhood of the current state s to find an action a (such that $a \in A(s)$) to reach the given target. In our current experiments, we use a real-time heuristic search called real-time A* (Korf, 1990) as a LPM with no modification of the heuristics (i.e. without learning). The pseudo-code of S*imulateAction* is the same as used in UCT (given in figure 2).

Related Work

A variation of UCT has been applied in a RTS game by (Balla and Fern, 2009) for a tactical assault problem. Balla and Fern used the concept of objective functions to measure the reward of the state-action pair. For example, a state-action pair is granted a high reward value if the leaf node (of the current rollout) reduces the time to attack or maximizes the player's health. However, only one objective (either minimum attack time or maximum health) can be used in the UCT planner at a time. Balla and Fern removed the C_p constant from the UCT exploration and exploitation balancing equation (1) and use the number of times an action is taken in a state to adjust the exploration and exploitation trade-off. This is the main variation Balla and Fern made in the UCT planner. This is also a difference between our work and that of Balla and Fern.

The other path finding methods which have been commonly used in computer games are A* (Hart et al., 1968), Navigational Mesh (Tozour 2002 and Hamm, 2008) and Waypoints (Rabin, 2000). Since A* searches for a full path between two locations before executing the plan, it is not suitable for real-time dynamic gaming environments. The navigational mesh and waypoints based path finding is also not suitable for the RTS games because the whole map is not available to process before the start of the game and also due to the frequent changes in the topography of the game map. These frequent changes are made by the construction of new buildings or when the dynamic obstacles (e.g. tanks, trucks or ships) cease to move.

RRT has been extensively used in robot motion planning problems where it is used to identify collision free parts of the search space through a random sampling approach. The main benefits of RRT based path planning in computer games over commonly used path planning approaches (i.e. A*, navigational mesh and waypoints) are i) RRT can be used for path planning in high dimensional search spaces (e.g. 3 degrees of freedom or higher) ii) RRT can handle uncertainty and iii) RRT can be used to generate waypoints dynamically in an online planning process. The disadvantage of RRT path planning is its poor performance in spaces with narrow passages (Zhang and Manocha, 2008).

Experimental Design

Experiments are performed on a desktop computer with 3.0 GHz computing speed and 1.0 GB RAM. We use three maps of 50x50 tiles for the experiments. We use simulation length (number of rollouts) as the stopping condition for the UCT and RRT planners. We use five different simulation lengths which are 30, 60, 90, 120 and 200. The UCT parameter C_p is set to 0.1 and the look-ahead depth is set to 4 for all experiments. A state in S is represented by (x, y) coordinates. The actions in the Action set are the pairs (dx, dy) where dx = {-1, 0, 1} and dy = {-1, 0, 1}. The action (0, 0) is excluded from the action set because path planning is always initiated when a character is in a stop state. An action does not bring a change in the state of a character if it is in front of a dynamic or static obstacle. A tank also becomes a static obstacle when it ceases to move. The unseen dynamic and static obstacles introduce uncertainty into the planning domain. A map is studied with three problems, where each problem has different coordinates (in terms of control centre and mineral cluster) to the others. In each map, the numbers of dynamic and static obstacles, and their placement coordinates, are kept different to other maps.

The performance of UCT and RRT planners is measured using four evaluation parameters: Score, Time, Planning cost and Convergence cost. Time is measured in the number of seconds that a method takes to play a game of 10K frames. The Scoring mechanism has already been described in section "RC-RTS Game" and it represents the total number of times the targets are achieved. Planning cost is the number of states seen during the online planning episodes. Convergence cost is the number of actions sampled during the online planning work.

Results

The results presented in this section are the average of the 15 runs for each simulation length and map. These average values are presented with the standard error of the means. A comparison of the UCT and RRT planners, with respect to convergence cost, is given in figure 5.

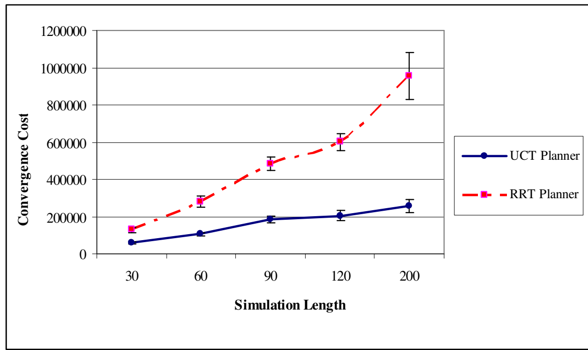

Figure 5: Convergence cost versus simulation length

UCT performs significantly better than RRT in terms of convergence cost. This is due to UCT's selective action sampling scheme (the balance between exploration and exploitation). The RRT planner's scheme of action selection, at a state, depends on the neighbours of that state and a random state (part of the state space but not in the look-ahead tree). Therefore, it explores a large number of actions for all simulation lengths. The convergence cost in RRT planner increases exponentially with the increase in the simulation length.

The planning cost for all five simulation lengths is shown in figure 6. These results show a huge difference between the UCT and RRT planners with respect to the length of the exploration of the state space. UCT looks at significantly fewer states to find the solution when compared to the RRT planner. This huge gap is mainly due to the difference in the schemes of action selection. RRT planner's rollout search focuses on finding a new neighbouring state of a given state in each rollout (i.e. exploration); therefore, it expands the state space search to a very large extent while UCT's lower searching efforts than RRT are due to its ability to adjust the trade-off between exploration and exploitation.

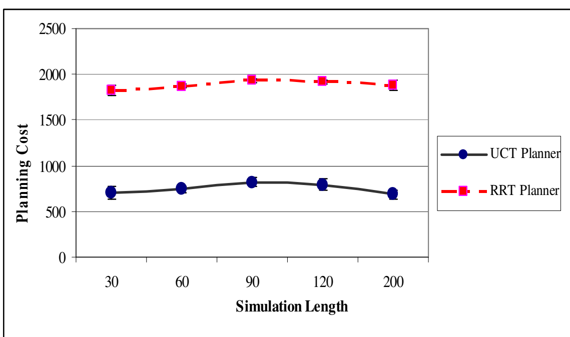

Figure 6: Planning cost versus simulation length

Figure 7 gives a graphical view of the performance of both planners with respect to the mean score. These results show that RRT performs better than UCT in terms of game playing intelligence. The better performance of RRT shows the importance of its random sampling based exploration capabilities. The common feature between both planners is a decrease in average score with higher simulation lengths; i.e. 120 and 200. A visual display of these games shows that the workers mostly do the searching for the paths in the areas of the map with no obstacles, even when these areas are very far from the target location. This is due to the collision avoidance factor $|S_s|$. With the large number of rollouts, the reward function becomes biased towards $|S_s|$ and ignores the distance heuristics.

The game play performance shown in figure 7 can also be discussed using the minimum and maximum scores achieved by both planners in each simulation length. These scores are shown in figure 8. RRT planner achieved higher maximum scores than UCT planner with every simulation length. The best score (the highest maximum score) of UCT planner in all simulation lengths is 320 which is approximately 30% smaller than the best for the RRT planner. There is no major change in the RRT planner's minimum score with respect to the change in simulation length. The UCT planner's minimum depends on the simulation length and it has higher minimum scores with higher simulation lengths than that of smaller simulation lengths.

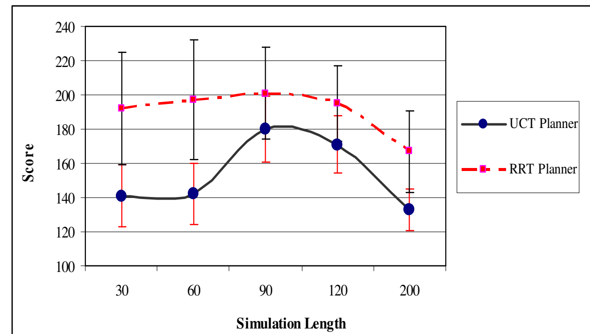

Figure 7: Mean score versus simulation length

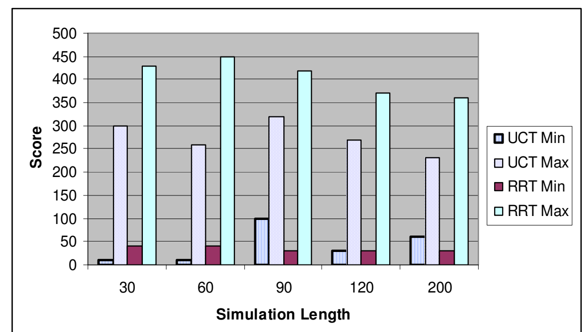

Figure 8: Min and max scores versus simulation length

Time durations spent by both planners to play games with different simulation lengths are given in figure 9. The UCT planner plans paths significantly quicker than the RRT

planner with a simulation length 60; otherwise there is no significant difference between the planners with respect to the time usage. However, the time usage profile falls at the simulations lengths 120 and 200. This is a result of the workers' movements in the obstacle free areas due to the reward function's convergence into the space of high $|S_s|$. In the obstacle-free area, the workers do not demand replanning frequently and as such saves planning time.

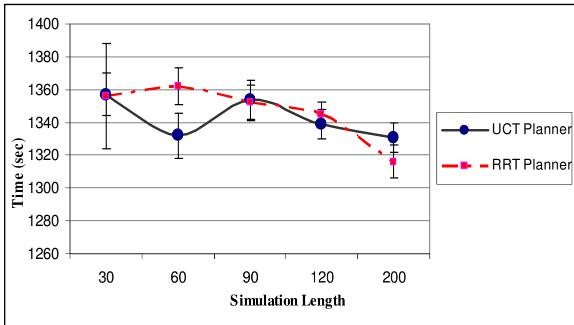

Figure 9: Time versus simulation length

Discussion

The performance of two Monte-Carlo planners, UCT and RRT, is evaluated in a RTS game for solving the path finding problem. Both planners are run within a RTS game with a fixed number of game ticks. The results show that UCT finds solutions with less search effort than the RRT planner. The RRT planner performs better than UCT in terms of game play when run with a specific simulation length. The results also provide a potentially useful insight into the application of Monte-Carlo planning in a RTS game for solving the path planning problem.

As this is a preliminary work, there are several promising directions which are possible for future work. One of them is the modification of the reward function to adjust the parameters: distance to goal and collision estimation. We plan to use a weighted scheme to avoid the possible biasing in the reward function towards one parameter when the rollouts are increased. We also aim to modify the reward function by adding an estimation of the collision with dynamic obstacles.

The UCT parameter C_p also needs more investigation. In the current set of experiments, we arbitrarily selected a small value of this parameter. We plan to study the impact of C_p on the performance of UCT if its value is increased from a small amount to a larger amount (i.e., $C_p = 1$). We also aim to investigate the possible relationship between C_p and the domain.

The RRT planner will be extended to generate and maintain an explicit tree of collision free nodes. This tree can be used as a set of way points. The random sampling scheme of RRT planner can be modified to generate random samples according to the goal locations. This can reduce the searching efforts of the planner. We also aim to extend path planning in RC-RTS for the tanks. We plan to modify the definition of the game to include the enemy units. The higher values of the standard errors of the means in the results of both planners also suggest running large numbers of test games for each parameter (i.e. considerably larger than 15).

Since the game world is represented as the 2D grid, it is also interesting to apply the graph search techniques in RC-RTS for the path planning. These techniques include the variations of A* (which are suitable for the real-time systems) and Bellman equation based asynchronous dynamic programming (e.g. Barto *et al*., 1991).

Acknowledgements

This research is in part supported by the University of Huddersfield, School of Computing and Engineering. We are thankful to the ORTS development team for providing a free tool for researching the RTS games. We borrow some pieces of code given in ORTS example projects. These include an event handler from "SampleAI" project, ridge creation from "Game-1" and the game map representation scheme from "Simple Pathfinding" project.

References

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

Barto, A.G., Bradtke, S.J., and Singh, S.P. 1991. Real-time Learning and control using Asynchronous Dynamic Programming. *Technical Report 91-57*, Computer Science Department, University of Massachusetts.

Bjarnason, R., Fern, A., and Tadepalli, P. 2009. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *proceedings of the ICAPS-2009*, 26-33.

Bulitko, V., and Lee, G. 2006. Learning in Real-Time Search: a Unifying Framework. *Journal of Artificial Intelligence Research (JAIR)*, 25(1): 119-157.

Buro, M. 2002. ORTS: A Hack-free RTS Game Environment. In *proceedings of the International Computers and Games Conference*, 280-291.

Chung, M., Buro, M., and Shaeffer, J. 2005. Monte-Carlo Planning in RTS games. In *proceedings of the Computational Intelligence and Games 2005*, UK.

Hamm, D. 2008. Navigational Mesh Generation: An empirical Approach. In *S. Rabin (Ed.), AI Game*

Programming Wisdom 4, Hingham, MA: Charles River Media Publisher, 113-114.

Hart, P.E., Nilsson, N.J., and 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.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *proceedings of 15th European Conference on Machine Learning*, 282-293.

Korf, R. 1990. Real-Time Heuristic Search. *Artificial Intelligence*, 42(2-3): 189-211.

Kuffner, J.J. and LaValle, S.M. 2000. RRT-Connect: An Efficient Approach to Single-Query Path Planning. In *proceedings of IEEE International Conference on Robotics and Automation (ICRA 2000)*, 995-1001.

Lee, C-S., Wang, M-H., Chaslot, G., Hoock, J-B., Rimmel, A., Teytaud, O., Tsai, S-R., Hsu, S-C., and Hong, T-P. 2009. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1): 73-89.

LaValle, S.M. 2006. *Planning Algorithms*. New York: Cambridge University Press.

Naveed, M., Kitchin, D., and Crampton, A. 2010. A Hierarchical Task Network Planner for Pathfinding in Real-Time Strategy Games. In *Proceedings of the Third International Symposium on AI & Games, Daniela M. Romano and David C. Moffat (Eds.)*, AISB 2010, 1-7.

Péret, L., and Garcia, F. 2004 . On-line search for solving Markov Decision Processes via heuristic sampling. In *proceedings of the 16th European Conference on Artificial Intelligence*, 530-534.

Rabin, S. 2000. A* speed optimizations. In *M. Deloura (Ed.), Game Programming Gems*, Hingham, MA: Charles River Media Publisher.

Tozour, P. 2002. Building a near-optimal navigational mesh. In *S. Rabin (Ed.), AI Game Programming Wisdom*, Hingham, MA: Charles River Media Publisher, 171-185.

Vanderbei, R. 1996. Sailing Strategies: An Application involving stochastics, Optimization, and Statistics (SOS). *http://orfe.princeton.edu/~rvdb/sail/sail.html*, AccessDate:13 Sept 2010.

Zhang, L., and Manocha, D. 2008. An Efficient Retraction-based RRT Planner. In *proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2008),* 3743 - 3750.