Monte Carlo Planning in RTS Games

Michael Chung, Michael Buro, and Jonathan Schaeffer
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{mchung,mburo,jonathan}@cs.ualberta.ca

Abstract- Monte Carlo simulations have been successfully used in classic turn–based games such as backgammon, bridge, poker, and Scrabble. In this paper, we apply the ideas to the problem of planning in games with imperfect information, stochasticity, and simultaneous moves. The domain we consider is real–time strategy games. We present a framework — MCPlan — for Monte Carlo planning, identify its performance parameters, and analyze the results of an implementation in a capture–the–flag game.

1 Introduction

Real–time strategy (RTS) games are popular commercial computer games involving a fight for domination between opposing armies. In these games, there is no notion of whose turn it is to move. Both players move at their own pace, even simultaneously; delays in moving will be quickly punished. Each side tries to acquire resources, use them to gain information and armaments, engage the enemy, and battle for victory. The games are typically fast–paced and involve both short–term and long–term strategies. The games are well–suited to Internet play. Many players prefer to play against human opponents over the Internet, rather than play against the usually limited abilities of the computer artificial intelligence (AI). Popular examples of RTS games include WarCraft [1] and Age of Empires [2].

The AI in RTS games is usually achieved using scripting. Over the past few years, scripting has become the most popular representation used for expressing character behaviours. Scripting, however, has serious limitations. It requires human experts to define, write, and test the scripts comprised of 10s, even 100s, of thousands of lines of code. Further, the AI can only do what it is scripted to do, resulting in predictable and inflexible play. The general level of play of RTS AI players is weak. To enable the AI to be competitive, game designers often give AI access to information that it should not have or increase its resource flow.

Success in RTS games revolves around planning in various areas such as resource allocation, force deployment, and battle tactics. The planning tasks in an RTS game can be divided into three areas, representing different levels of abstraction:

1. **Unit control (unit micromanagement)**. At the lowest level is the individual unit. It has a default behaviour, but the player can override it. For example, a player may micromanage units to improve their performance in battle by focusing fire to kill off individual enemy units.

2. **Tactical planning (mid–level combat planning).** At this level, the player decides how to conduct an attack on an enemy position. For example, it may be possible to gain an advantage by splitting up into two groups and simultaneously attacking from two sides.

3. **Strategic planning (high–level planning).** This includes common high–level decisions such as when to build up the army, what units to build, when to attack, what to upgrade, and how to expand into areas with more resources.

In addition, there are other non–strategic planning issues that need to be addressed, such as pathfinding.

Unit control problems can often be handled by simple reactive systems implemented as list of rules, finite state machines, neural networks, etc. Tactical and strategic planning problems are more complicated. They are real–time planning problems with many states to consider in the absence of perfect information. It is apparent that current commercial RTS games deal with this in a simple manner. All of the AI's strategies in the major RTS games are scripted. While the scripts can be quite complex, with many random events and conditional statements, all the strategies are still predefined beforehand. This limitation results in AI players that are predictable and thus easily beaten. For casual players, this might be fun at first, but there is no re–playability. It is just no fun to beat an AI player the same way over and over again.

In RTS games, there are often hundreds of units that can all move at the same time. RTS games are fast–paced, and the computer player must be able to make decisions at the same speed as a human player. At any point in time, there are many possible actions that can be taken. Human players are able to quickly decide which actions are reasonable, but current state–of–the–art AI players cannot. In addition, players are faced with imperfect information, i.e. partial observability of the game state. For instance, the location of enemy forces is initially unknown. It is up to the players to scout to gather intelligence, and act accordingly based on their available information. This is unlike the classical games such as chess, where the state is always completely known to both players. For these reasons, heuristic search by itself is not enough to reason effectively in an RTS game. For planning purposes, it is simply infeasible for the AI to think in terms of individual actions. Is there a better way?

Monte Carlo simulations have the advantage of simplicity, reducing the amount of expert knowledge required to achieve high performance. They have been successfully used in games with imperfect information and/or stochastic elements such as backgammon [14], bridge [9], poker [5], and Scrabble [11]. Recently, this approach has been tried in two-player perfect-information games with some success (Go [6]). A framework for using simulations in a game–playing program is discussed in [10], and the subtleties of getting the best results with the smallest sample sizes is discussed in [12].

Can Monte Carlo simulations be used for planning in RTS games? If so, then the advantages are obvious. Using simulations would reduce the reliance on scripting, resulting in substantial savings in program development time. As well, the simulations will have no or limited expert bias, allowing the simulations to explore possibilities not covered by expert scripting. The result could be a stronger AI for RTS games and a richer gaming experience.

The contributions in this work are as follows:

1. Design of a Monte Carlo search engine for planning (MCPlan) in domains with imperfect information, stochasticity, and simultaneous moves.
2. Implementation of the MCPlan algorithm for decision making in a real–time capture–the–flag game.
3. Characterization of MCPlan performance parameters.

Section 2 describes the MCPlan algorithm and the parameters that influence its performance. Section 3 discusses the implementation of MCPlan in a real–time strategy game built on top of the free ORTS RTS game engine [7]. Section 4 presents experimental results. We finish the paper by conclusions and remarks on future work in this area.

2 Monte Carlo Planning

Adversarial planning in imperfect information games with a large number of move alternatives, stochasticity, and many hidden state attributes is very challenging. Further complicating the issue is that many games are played with more than two players. As a result, applying traditional game–tree search algorithms designed for perfect information games that act on the raw state representation is infeasible. One way to make look–ahead search work is to abstract the state space. An approach to deal with imperfect information scenarios is sampling. The technique we present here combines both ideas.

Monte Carlo sampling has been effective in stochastic and imperfect information games with alternating moves, such as bridge, poker, and Scrabble. Here, we want to apply this technique to the problem of high–level strategic planning in RTS games. Applying it to lower level planning is possible as well. The impact of individual moves — such as a unit moving one square — requires a very deep search to see the consequences of the moves. Doing the search at a higher level of abstraction, where the execution of plan becomes a single "move", allows the program to envision the consequences of actions much further into the future (see Section 2.2).

Monte Carlo planning (MCPlan) does a stochastic sample of the possible plans for a player and selects the plan to execute that has the highest statistical outcome. The advantage of this approach is that it reduces the amount of expert–defined knowledge required. For example, Full Spectrum Command [3] requires extensive military–strategist–defined plans that the program uses — essentially forming an expert system. Each plan has to be fully specified, including identifying the scenarios when the plan is applicable, anticipating all possible opponent reactions, and the consequences of those reactions. It is difficult to get an expert's time to define the plans in precise detail, and more difficult to invest the time to analyze them to identify weaknesses, omissions, exceptions, etc. MCPlan assumes the existence of a few basic plans (e.g. explore, attack, move towards a goal) which are application dependent, and then uses sampling to evaluate them. The search can sample the plans with different parameters (e.g. where to attack, where to explore) and sequences of plans—for both sides. In this section, we describe MCPlan in an application–independent manner, leaving the application–dependent nuances of the algorithm to Section 3.

2.1 Top–Level Search

The basic high–level view of MCPlan is as follows, with a more formal description given in Figure 1:

1. Randomly generate a plan for the AI player.
2. Simulate randomly–generated plans for both players and execute them, evaluate the game state at the end of the sequence, and compute how well the selected plan seems to be doing (`evaluate_plan`, Section 2.3).
3. Record the result of executing the plan for the AI player.
4. Repeat the above as often as possible given the resource constraints.
5. Choose the plan for the AI player that has the best statistical result.

The variables and routines used in Figure 1 are described in subsequent subsections.

The top–level of the algorithm is a search through the generated plans, looking for the one with the highest evaluation. The problem then becomes how best to generate and evaluate the plans.

2.2 Abstraction

Abstraction is necessary to produce a useful result and maintain an acceptable run–time. Although this work is discussed in the context of high–level plans, the implementor is free to choose an appropriate level of abstraction, even at the level of unit control, if desired. However, since MCPlan relies on the power of statistical sampling, many data points are usually needed to get a valid result. For best performance, it is important that the abstraction level be chosen to make the searches fast and useful.

In Figure 1, `State` represents an abstraction of the current game state. The level of abstraction is arbitrary, and in simple domains it may even be the full state.

2.3 Evaluation Function

As in traditional game–playing algorithms, at the end of a move sequence an evaluation function is called to assess how good or bad the state is for the side to move. This typically requires expert knowledge although the weight or

```
// Plan: contains details about the plan
// For example, a list of actions to take
class Plan {
  // returns true if no actions remaining in the plan
  bool is_completed();
  // [...] (domain specific)
};

// State: AI's knowledge of the state of the world
class State {
  // return evaluation of the current state
  // (domain specific implementation)
  float eval();
  // [...] (domain specific)
};

// MCPlan Top-Level
Plan MCPlan(
  State state,      // current state of the world
  int num_plans,    // number of plans to evaluate
  int num_sims,     // simulations per evaluation
  int max_t)        // max time steps per simulation
{
  float best_val = -infinity;
  Plan best_plan;

  for (int i = 0; i < num_plans; i++) {
    // generate plan using (domain-specific) plan generator
    Plan plan = generate_plan(state);
    // evaluate using the number of simulations specified
    float val = evaluate_plan(plan, state, num_sims, max_t);
    // keep plan with the best evaluation
    if (val > best_val) {
      best_plan = plan;
      best_val = val;
    }
  }
  return best_plan;
}
```

Figure 1: MCPlan: top–level search

importance of each piece of expert knowledge can be evaluated automatically, for example by using temporal difference learning [13]. For most application domains, including RTS games, there is no easy way around this dependence on an expert. Note that, unlike scripted AI which requires a precise specification and extensive testing to identify omissions, evaluation functions need only give a heuristic value.

2.4 Plan Evaluation

Before we describe the search algorithm in more detail, let us define the key search parameters. These are variables that may be adjusted to modify the quality of the search, as well as the run–time required. The meaning of these parameters will become more clear as the search algorithm is described.

1. `max_t`: the maximum time, in steps or moves, to look ahead when performing the simulation–based evaluation.
2. `num_plans`: the total number of plans to randomly generate and evaluate at the top–level.
3. `num_sims`: the number of move sequences to be considered for each plan.

The `evaluate_plan()` function is shown in Figure 2. Each plan is evaluated `num_sims` times. A plan is evaluated using `simulate_plan()` by executing a series of plans for both sides and then using an evaluation function to assess the resulting state. In the pseudo–code given, the value of a plan is the minimum of the sample values (a pessimistic assessment). Other metrics are possible, such as

```
// Evaluate Plan Function. Takes minimum of num_sims
// plan simulations (pessimistic)
float evaluate_plan(Plan plan, State state,
                    int num_sims, int max_t)
{
  float min = infinity;
  for (int i = 0; i < num_sims; i++) {
    float val = simulate_plan(plan, state, max_t);
    if (val < min) min = val;
  }
  return min;
}
```

Figure 2: MCPlan: plan evaluation

```
// Simulate Plan. Perform a single simulation with the given
// plan and return the resulting state's evaluation.
float simulate_plan(Plan plan, State state, int max_t)
{
  State bd_think = state;
  Plan plan_think = plan;

  // generate a plan for the opponent (domain specific)
  Plan opponent_plan = generate_opponent_plan(state);

  while (true) {
    // simulate a single time step in the world
    // (domain specific)
    simulate_plan_step(plan_think, opponent_plan, bd_think);

    // check if maximum time steps has been simulated
    if (--max_t <= 0) break;

    // check if plan has been completed
    if (plan_think.is_completed()) break;

    // check if the opponent's plan has been completed
    if (opponent_plan.is_completed()) {
      // if so, generate a new opponent plan
      opponent_plan = generate_opponent_plan(bd_think);
    }
  }
  return bd_think.eval();
}
```

Figure 3: MCPlan: plan simulation

taking the maximum over all samples, the average of the samples, or a function of the distribution of values. Also, in the presented formulation of MCPlan information about the plan chosen by the player is implicitly leaking to the opponent. This turns a possible imperfect information scenario into one of perfect information leading to known problems [8]. We will address this problem in future work. Here, we restrict ourselves to a simple form which nevertheless may be adequate for many applications. Each data point for a plan evaluation is done using `simulate_plan()`. Both sides select a plan and then executes it. This is repeated until time runs out. The resulting state of the game is assessed using the evaluation function. Note that opponent plans can cause interaction; how this is handled is application dependent and it is discussed in Section 3. The `evaluate_plan()` function calls `simulate_plan()` `num_sims` times, and takes the minimum value. Figure 3 shows the `simulate_plan()` function.

2.5 Comments

MCPlan is similar to the stochastic sampling techniques used for other games. The fundamental difference — besides obvious semantic ones such as not requiring players to alternate moves — is that the "moves" can be executed at an abstract level. Abstraction is key to getting the depths of search needed to have long–range vision in RTS games.

MCPlan lessens the dependence on expert–defined knowledge and scripts. Expert knowledge is needed in two places:

1. Plan definitions. A plan can be as simple or as detailed as one wants. In our experience, using plan *building blocks* is an effective technique. Detailed plans are usually composed of a series of repeated high–level actions. By giving MCPlan these actions and allowing it to combine them in random ways, the program can exhibit subtle and creative behaviour.

2. Evaluation function. Constructing accurate evaluation functions for non–trivial domains requires expert knowledge. In the presence of look–ahead search, however, the quality requirements can often be lessened by considering the well–known trade–off between search and knowledge. A good example is chess evaluation functions, which — combined with deep search — lead to World–class performance, in spite of the fact that the used features have been created by programmers rather than chess grandmasters. Because RTS games have much in common with classical games, we expect a similar relationship between evaluation quality and search effort in this domain, thus mitigating the dependency on domain experts.

3 Capture the Flag

Commercial RTS games are complex. There are many different variations, some involving many RTS game elements such as resource gathering, technology trees, and more. To more thoroughly evaluate our RTS planners, we limit our tests to a single RTS scenario, capture–the–flag (CTF). Our CTF game takes place on a symmetric map, with vertical and horizontal walls. The two forces start at opposing ends of the map. Initially the enemy locations are unknown. The enemy flag's location is known — otherwise much initial exploration would be required. This is consistent with most commercial RTS games, where the same maps are used repeatedly, and the possible enemy locations are known in advance.

The rules of our CTF game are relatively simple. Each side starts with a small fixed number of units, located near a home base (post), and a flag. Units have a range in which they can attack an opponent. A successful attack reduces the nearby enemy unit's hit–points. When a unit's hit–points drops to zero, the unit is "dead" and removed from the game.

The objective of CTF is to capture the opponent's flag. Each unit has the ability to pick–up or drop the enemy flag. To win the game, the flag must be picked up, carried, and dropped at the friendly home base. If a unit is killed while carrying the flag, the flag is dropped at the unit's location, and can later be picked up by another unit. A unit cannot pick up its own side's flag at any time.

Terrain is very important to CTF. For most of our tests we keep it simple and symmetric to avoid bias towards either side. However, even with more complex terrains, while there may be a bias towards one side, it is expected that planners that perform better on symmetric maps will also perform better on complex maps. While CTF does not capture all the elements involved in a full RTS game — such as economy and army–building — it is a good scenario for testing planning algorithms. Many of the features of full RTS games are present in CTF — including scouting and base defense.

Before we discuss how we applied MCPlan to a CTF game we first describe the simulation software we used.

3.1 ORTS

ORTS (Open RTS) is a free software RTS game engine which is being developed at the University of Alberta and licensed under the GNU General Public License. The goal of the project is to provide AI researchers and hobbyists with an RTS game engine that simplifies the development of AI systems in the popular commercial RTS game domain. ORTS implements a server–client architecture that makes it immune to map–revealing client hacks which are a widespread plague in commercial RTS games. ORTS allows users to connect *whatever* client software they wish — ranging from distributed RTS game AI to feature–rich graphics clients. The CTF game which we use for studying MCPlan performance has been implemented within the ORTS framework. For more information on the status and development of ORTS we refer readers to [4][7].

3.2 CTF Game State Abstraction

In the state representation, the map is broken up into tiles (representing a set of possible unit locations). Units are located on these tiles, and their positions are reasoned about in terms of tiles, rather than exact game coordinates. The state also contains information about the units' hit–points, as well as locations of walls and flags.

3.3 Evaluation Function

We tried to keep our evaluation function simple and obvious, without relying on a lot of expert knowledge. The evaluation function for our CTF AI has three primary components: material, exploration/visibility, and capture/safety. The first two components are standard to any RTS game. The third component is specific to our CTF scenario. Without it, the AI would have no way to know that it was actually playing a CTF game, and it would behave as if it was a regular battle. In each component the difference of the values for both players is computed. In the following we briefly give details of the evaluation function.

Material
The most important part of any RTS game is material. In most cases, the side with the most resources — including military units, buildings, etc. — is the victor. Thus, maximizing material advantage is a good sub–goal for any planning AI. This material can later be converted into a decisive advantage such as having a big enough army to eliminate the enemy base. There is a question of how to compare healthy units to those with low hit–points. For example, while it may be clear that two units each with 50% health are better than one unit with 100% health, which would be better, one unit with 100% health, or two units with 25% health? While the two units could provide more firepower, they could also

be more quickly killed by the enemy. There are different situations where the values of these units may be different. For our tests, we provide a simple solution: each unit provides a bonus of $\sqrt{0.01 \times \text{hp}}$. The maximum hp (hit–point) value is 100. Thus, each live unit has a value of between 0.1 and 1. The value for friendly units is added to our evaluation, and enemy units values are subtracted. Taking the square root prefers states which — for a constant hit–point total — have a more balanced hit–point distribution.

Exploration and Visibility
When not doing something of immediate importance, such as fighting, exploring the map is very important. The side with more information has a definite advantage. Keeping tabs on the enemy, finding out the lay of the land, and discovering the location of obstacles are all important. The planner cannot accurately evaluate its plans unless it has a good knowledge of the terrain and of enemy forces and their locations. The value of information is reflected by these evaluation function sub–components:

- Exploration bonus: $0.001\times$ # of explored tiles, and
- Vision bonus: $0.0001\times$ # visible tiles.

Note that the bonus values can be changed or even learned.

Flag Capture and Safety
To win a CTF game, the opponent's flag has to be captured. It is important to encourage the program to go after the enemy's flag, while at the same time ensuring that the program's flag remains safe:

- Bonus for being close to enemy flag: +0.1 per tile,
- Bonus for possession of enemy flag: +1.0,
- Bonus for bringing enemy flag closer to our base: +0.2 per tile, and
- Similar penalties apply if the enemy meets these conditions.

Note that all these heuristic values have been manually tuned. Machine learning would be a way to more reliably set these values.

Combining the Components
The simplest thing to do, and what we do right now, is have constant factors for adding the three components together. There are exceptions where this is not the best approach. For example, if we are really close to capturing the enemy flag, we may choose to ignore the other components, such as exploration. Such enhancements are left as future work. In our experiments we give each component equal weight.

Evaluation Function Quality
We can perform experiments to test the effectiveness of our evaluation function. For example, we could measure the time it takes to capture the flag if there are no enemy units. This removes all tactical situations and focuses on testing that the evaluation function is correctly geared towards capturing the enemy flag. Playing the MCPlan AI against a completely random AI also provides a good initial test of the evaluation function. A random evaluation function would perform on the same level as the random AI, whereas a better evaluation function would win more often.

3.4 Plan Generation

There are two types of plan generation used in this project: random and scripted. The random plans are simple and are described below. The scripted plans are slightly more sophisticated, but still quite simple. Only the random plans are used in this implementation, as we do not have many scripted plans implemented.

Random Plans
A random plan consists of assigning a random nearby destination for each unit to move to. That is, for each unit, a nearby unoccupied destination tile is selected. The maximum distance to the destination is determined by the `max_dist` variable. The A* pathfinding algorithm is then used to find a path for each unit. Note that collisions are possible between the units, but are ignored for planning purposes. We did not implement any group–based pathfinding, although it is a possible enhancement.

Scripted Plans
We have implemented a small number of action scripts which provide test opponents for the MCPlan algorithm. As previously mentioned, scripted plans have many disadvantages — most notably, the need to have an expert define, refine and test them. However, there is the possibility that given a set of scripted plans, applying the search and simulation algorithms described in this paper can result in a stronger player.

3.5 Plan Step Simulation

Simulation must be used because when the planner evaluates an action, the result of that action cannot be perfectly determined because of hidden enemy units, unknown enemy actions, randomized action effects, etc. Also, as our simulation acts on an abstracted state description, the computation should be much faster. The plan step simulation function takes the given plans for the friend and enemy sides and executes one–tile moves for each side. Unit attacks are then simulated by selecting the nearest opposing unit for each unit, and reducing its hit–points. The attacks may not match what would happen in the actual game, due to many reasons. For example, units may seem to be in range but actually they are not, due to the abstracted distances. Also, in some games, the attack damage is random, so the damage results may not be exactly the same as what will happen in the game. However, it is expected that with a large enough value of `num_evals`, the final result should be more statistically accurate.

3.6 Other Issues

In this subsection we discuss some implementation issues related to developing and testing a search/simulation based RTS planning algorithm such as MCPlan.

Map Generation
It is clear that in performing the tests, map generation is a hard problem. To produce an unbiased map, the map should be completely symmetric. A more complex asymmetric map could favor one side. In addition, it is possible that

different types of maps could favour different AI's. For our tests we use a simple symmetric map, to avoid most of these issues. It is expected — and to be confirmed — that on more complex and on randomly generated maps, the conclusions we draw from our experiments should still hold.

Server Synchronization
The tests should be run with server synchronization turned on. This option tells the ORTS server to wait for replies from both clients before continuing on to the next turn. In the default mode with synchronization off, the first player to connect may possibly have an advantage, due to being able to move while the second player's process is still initializing its GUI, etc. The server synchronization option eliminates this possible source of bias, as well as reducing the randomness caused by random network lag.

Interactions and Replanning
As players interact previous planning may quickly become irrelevant. In many cases, replanning must occur. Not every interaction should result in replanning. This would result in too frequent replanning, which would slow down the computation while perhaps not improving the decision quality much. Instead, only important interactions should result in replanning. Possible such interactions are: "a unit is destroyed," "a unit is discovered," or "a flag is picked up." Note that attacks, while important, happen too frequently and thus should not trigger replanning.

4 Experiments

In this section, we investigate the performance issues of MCPlan on our CTF game.

4.1 Experimental Design

Each experimental data point consisted of a series of games between two CTF programs. The experiments were performed on 1.2 GHz PCs with 1 GB of RAM. Note that because the experiments were synchronized by the ORTS server the speed of the computer does not affect the results. Each data point is based on the results of matching two programs against each other for 200 games. For a given map, two games are played with the programs playing each side once. A game ends when a flag is captured, or one side has all their men eliminated. A point is awarded to the winning side. Draws are handled depending on the type of draw. If the game times out and there is no winner, then neither side gets a point. If both sides achieve victory at exactly the same time, then both sides get a point. The reported win percentage is one side's points divided by the total points awarded in that match. In a match with no draws the total points is equal to the number of games (200).

Maps
Figure 4 shows the maps that have been used in the experiments. Their dimensions are 20 by 20 tiles. By default each side starts with five men.

Search Parameters
The `max_dist` parameter is the maximum distance that a unit can move from its current position in a randomly generated plan. In all these experiments, the `max_dist` parameter is set to 6 tiles, unless otherwise stated. The unit's sight radius is set to 10 tiles, and unit's attack range is set to 5 tiles. To reduce the number of experiments needed, the number of simulations (`num_sims`) is set to be equal to the number of plans (`num_plans`). This makes sense as the number of simulations is also the number of opponent plans considered.

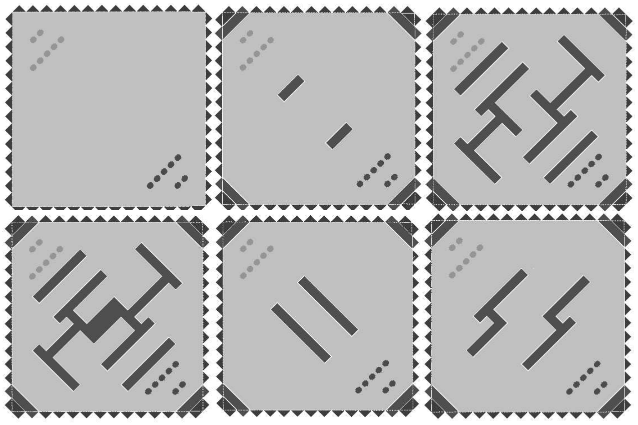
Figure 4: Maps and unit starting positions used in the experiments: **map 1** (upper left): empty terrain (this is the default), **map 2**: simple terrain with a couple of walls, **map 3**: complex terrain, **map 4**: complex terrain with dead–ends, **map 5**: simple terrain with a bottleneck, **map 6**: intermediate complexity.

Players
There are two opponents tested in these experiments other than the MCPlan player: Random and Rush–the–Flag. Random is equivalent to MCPlan running with `num_plans` = 1. It simply generates and executes a random plan, using the same plan generator as the MCPlan player. Rush–the–Flag is a scripted opponent which behaves as follows:

1. If the enemy flag is not yet captured, send all units towards the enemy flag and attempt to capture it.
2. If the enemy flag is captured, have the flag carrier return home. All other units follow the flag carrier.

While simple in design, the Rush–the–Flag opponent proves to be a strong adversary.

4.2 Results

We now investigate the performance of MCPlan against a variety of opponents and using different combinations of search parameters.

Increasing Number of Plans
In Figure 5, the performance of the MCPlan algorithm on the default map is evaluated as a function of the number of plans considered. Each data point represents the result of a player considering p plans playing against one that considers $2p$ plans. This results show that the program's play improves as the number of plans increases, but with diminishing returns. Eventually, the sample size is large enough that adding more plans results in marginal performance improvements, as expected.

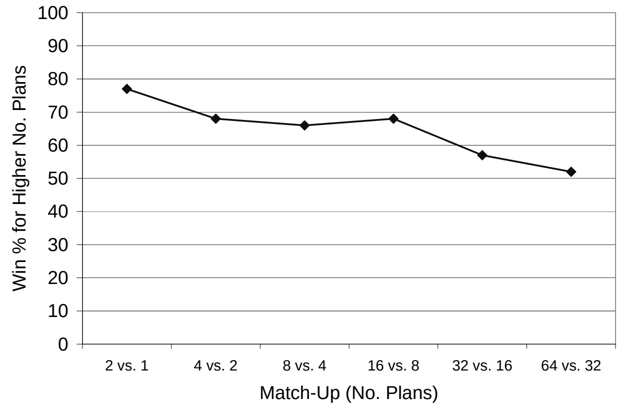

Figure 5: Increasing Number of Plans

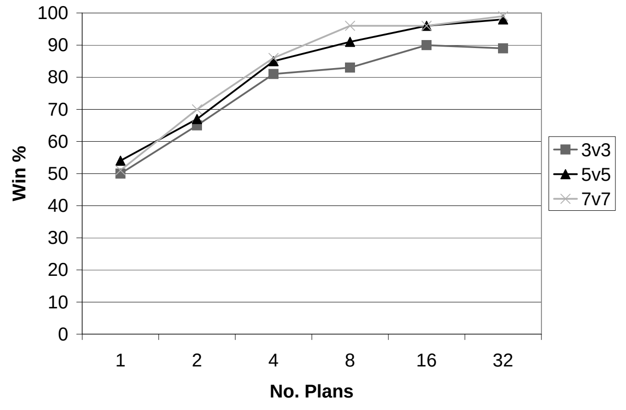

Figure 6: Different Number of Units. MCPlan vs. Random

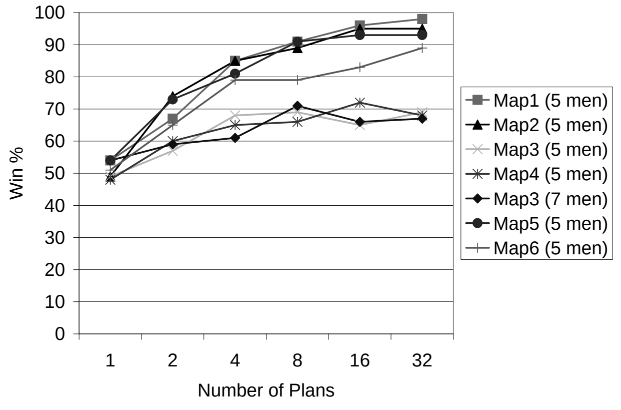

Figure 7: Different Maps. MCPlan vs. Random

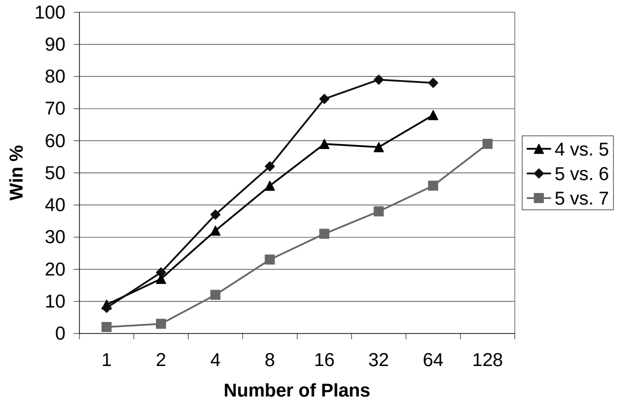

Figure 8: Less Men and Stronger AI vs. Random

Number of Units

Figure 6 shows the results when the number of units is varied. The results in the figure are for MCPlan against Random on the default map. As expected, regardless of the number of units aside, increasing the number of plans improves the performance of the MCPlan player. With a larger number of units per side, MCPlan wins more often. This is reasonable, as the number of decisions increases with the number of units, and there is more opportunity to make "smarter" moves.

Different Maps

The previous results were obtained using the same map. Do the results change significantly with different terrain? In this experiment, we repeat the previous matches using a variety of maps. Figure 7 shows the results. Note that one map has 7 men aside. The results indicate that MCPlan is a consistent winner, but the winning percentage depends on the map. The more complex the map, the better the random player performs. This is reasonable, since with more walls, there is more uncertainty as to where enemy units are located. This reduces the accuracy of the MCPlan player's simulations. In the tests using the map with a bottleneck (map 5), the performance was similar to the tests with simple maps without the bottleneck. This shows that the simulation is capable of dealing with bottlenecks, at least in simple cases.

Unbalanced Number of Units

Figure 8 illustrates the relative performance between MCPlan and Random when Random is given more men. The results show that given a sufficient number of plans to evaluate, MCPlan with less men and better AI can overcome Random with more men but a poorer AI. The results suggest that using MCPlan is strong enough to overcome a significant material advantage possessed by the weaker AI (Random). The figure shows the impressive result that 5 units with smart AI defeat 7 units with dumb AI 60% of the time when choosing between 128 plans.

Optimizing Max–Dist

A higher max_dist value results in longer plans, which allows more look–ahead, as well as a higher number of possible plans. The higher number of possible plans may increase the number of plans required to find a good plan.

More look–ahead should help performance. However, with too much look–ahead, noise may become a problem. The noise is due to errors in the simulation — which uses an abstracted game state — and incorrect predictions of the opponent plan. The longer we need to guess what the opponent will do, the more likely we are to make an error. So, more simulations are required to have a good chance of predicting the opponent's plan or something close enough to it.

In this experiment we vary the max_dist parameter to optimize the win percentage against the Random opponent on map 1 and the Rush–the–Flag opponent on map 2 (see Figure 9). The planner playing against random achieves its best performance of 94% at dist=6. Note that although one may expect MCPlan to score 100% against Random, in

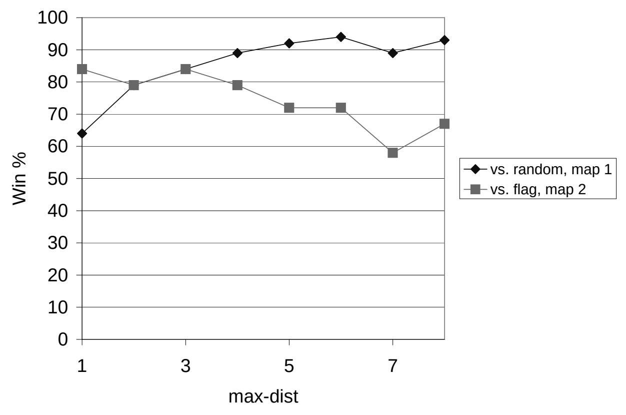

Figure 9: Optimizing Max–Dist Parameter

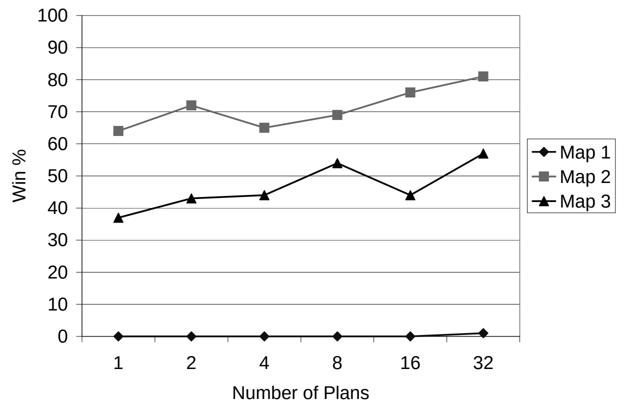

Figure 10: MCPlan vs. Rush–the–Flag Opponent

practice this will not happen. A lone unit may unexpectedly encounter a group of enemy units. Once engaged in a losing battle, it is difficult to retreat, since all units move at the same speed.

Rush–the–Flag Opponent

Figure 10 shows MCPlan playing against Rush–the–Flag. The playing strength of Rush–the–Flag is very map dependent, as it has a fixed strategy. On the first map, Rush–the–Flag wins nearly every game. Rushing is a near–optimal strategy on an empty map. On map 2, where the direct path to the other side is blocked, Rush–the–Flag is much weaker. MCPlan wins more than 60% of the time even with `num_plans` =1. With `num_plans` =32, MCPlan wins more than 80% of the time. However, on map 3, where the map is more complex and all paths to the other side are long, Rush–the–Flag again becomes a challenging opponent. However, with `num_plans` =32, MCPlan wins more than 55% of the games.

Run–Time for Experiments

In order to get more statistically valid results, the experiments were not run in real–time. Rather, they were run much faster than real–time, about 100 times faster.

While the run–time depends on the parameters, using typical parameters (map 1, 16 plans, 5 men per side) a 200–game match runs in about 80 minutes on our test machines. The average time per game is less than 30 seconds. As the planner re–plans hundreds of times per game, this results in planning times of a fraction of a second.

5 Conclusions and Future Work

This paper has presented preliminary work in the area of sampling–based planning in RTS games. We have described a plan selection algorithm – MCPlan – which is based on Monte Carlo sampling, simulations, and replanning. Applied to simple CTF scenarios MCPlan has shown promising initial results. To gauge the true potential of MCPlan we need to compare it against a highly tuned scripted AI, which was not available at the time of writing. We intend to extend MCPlan in various dimensions and apply it to more complex RTS games. For instance, it is natural to add knowledge about opponents in form of plans that can be incorporated in the simulation process to exploit possible weaknesses. Also, the top–level move decision routine of MCPlan should be enhanced to generate move distributions rather than single moves which is especially important in imperfect information games. Lastly, applying MCPlan to bigger RTS game scenarios requires us to consider more efficient sampling and abstraction methods.

Acknowledgments

We thank Markus Enzenberger and Nathan Sturtevant for valuable feedback on this paper. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

Bibliography

[1] http://www.blizzard.com .

[2] http://www.ensemblestudios.com .

[3] http://www.ict.usc.edu/disp.php?bd=proj_games_fsc1 .

[4] http://www.cs.ualberta.ca/~mburo/orts .

[5] D. Billings, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *AAAI National Conference*, pages 697–703, 1999.

[6] B. Bouzy and B. Helmstetter. Monte Carlo go developments. In *Advances in Computer Games X*, pages 159–174. Kluwer Academic Press, 2003.

[7] M. Buro and T. Furtak. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS), Arlington VA 2004*, pages 51–58, 2004.

[8] I. Frank and D.A. Basin. Search in games with incomplete information: A case study using bridge card play. *AI Journal*, 100(1-2):87–123, 1998.

[9] M. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*, pages 584–589, 1999.

[10] Jonathan Schaeffer, Darse Billings, Lourdes Peña, and Duane Szafron. Learning to Play Strong Poker. In J. Fürnkranz and M. Kubat, editors, *Machines That Learn To Play Games*, pages 225–242. Nova Science Publishers, 2001.

[11] B. Sheppard. *Towards Perfect Play in Scrabble*. PhD thesis, 2002.

[12] B. Sheppard. Efficient control of selective simulations. *Journal of the international Computer Games Association*, 27(2):67–80, 2004.

[13] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1998.

[14] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.