University of Alberta

Library Release Form

Name of Author: Frantisek Sailer

Title of Thesis: Adversarial Planning in RTS Games Through Simulation

Degree: Master of Science

Year this Degree Granted: 2007

_____

Frantisek Sailer

Date: _____

Life is "trying things to see if they work"

– Ray Bradbury

University of Alberta

ADVERSARIAL PLANNING IN RTS GAMES THROUGH SIMULATION

by

Frantisek Sailer

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Adversarial Planning in RTS Games Through Simulation** submitted by Frantisek Sailer in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Michael Buro

_____

Jonathan Schaeffer

_____

Petr Musilek

Date: _____

Abstract

Adversarial planning in complex decision domains, such as modern video games, has not yet received much attention from AI researchers.

This thesis presents a planning framework (RTSplan) that uses simulation combined with Nash-equilibrium strategy approximation to choose the best policy from a given policy set. We apply this framework to an army deployment problem in an abstract real-time strategy game setting. Experimental results indicate a performance gain over individual policies in our policy set. Furthermore, we show that adding basic opponent modelling drastically increases the performance of RTSplan against these policies, and that RTSplan can also play well against unknown policies.

We also present a method for the *fast-forwarding* of simulations which greatly reduces computation times.

RTSplan is an automated way of increasing the decision quality of scripted AI systems in real-time. It is suited for complex systems that have real-time constraints, simultaneous moves, and currently rely on scripted solutions.

Acknowledgements

Thanks to my supervisor Michael Buro for his endless patience and words of wisdom throughout this endeavour, and to Marc Lanctot for his valuable feedback. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

Table of Contents

1 Introduction **1**
- 1.1 Real-time Strategy Games . . . 1
- 1.2 RTS Game Strategic Areas . . . 2
- 1.3 Commercial RTS Game AI . . . 3
- 1.4 Computationally Difficult Properties of RTS Games . . . 4
- 1.5 Thesis Contributions . . . 6
- 1.6 Thesis Organization . . . 6

2 Planning in RTS Games **7**
- 2.1 Planning Overview . . . 7
- 2.2 Planning Methods Relevant to RTS Games . . . 9
  - 2.2.1 Hierarchical Task Networks . . . 9
  - 2.2.2 Monte Carlo Approach . . . 10
  - 2.2.3 Simulation-Based Planning . . . 12
- 2.3 AI in RTS Games . . . 13
  - 2.3.1 Map Analysis . . . 13
  - 2.3.2 Dependency Graphs . . . 15
  - 2.3.3 Multi-Tiered AI . . . 15
- 2.4 Conclusions . . . 16

3 Simulation-Based Adversarial Planning **18**
- 3.1 Action Abstraction . . . 18
- 3.2 Planning Based on Policy Simulation . . . 19
  - 3.2.1 The RTSplan Algorithm . . . 20
- 3.3 Opponent Modelling . . . 24

4 Implementation and Experiments **29**
- 4.1 Implementation Details . . . 29
  - 4.1.1 Fast-Forwarding of Policies . . . 36
- 4.2 Experimental Setup . . . 38
- 4.3 Initial Experiments . . . 39
  - 4.3.1 RTSplan Experimental Results . . . 41
  - 4.3.2 Results of RTSplan with Additional Policies . . . 45
  - 4.3.3 RTSplan-O Experimental Results . . . 48
  - 4.3.4 Execution Times . . . 53

5 Conclusions and Future Work **56**
- 5.1 Conclusions . . . 56
- 5.2 Future Work . . . 57

Bibliography **59**

List of Tables

4.1 Different Player Comparison 42
4.2 RTSplan Player vs. Specific Policies 44
4.3 Hunter vs. Specific Policies 44
4.4 RTSplan Player (no delay) vs. Specific Policies 44
4.5 Simulation Players Comparison 45
4.6 Harass vs. Specific Policies 47
4.7 Attack Least Defended vs. Specific Policies 47
4.8 RTSplan vs. New Policy Set 47
4.9 RTSplan-O vs. Policy Set ($t = 2.0, f = 0.5$) 49
4.10 RTSplan-O vs. Policy Set ($f = 0.1$) 49
4.11 RTSplan-O vs. Policy Set ($t = 2.0$) 49
4.12 RTSplan vs. Unknown Policy 51
4.13 RTSplan-O vs. Unknown Policy ($t = 2.0$, $f = 0.5$) 51
4.14 Opponent Modelling Policy Set Reduction ($t = 2.0$, $f = 0.5$, 50 maps) 55
4.15 Execution Times (milliseconds) Percentiles and Max Time 55
4.16 Fast-forwarding time comparison (seconds)(50 maps) 55

List of Figures

1.1 A typical Starcraft situation, where a base is attacked by enemy forces. 2

3.1 Top level planning cycle 20
3.2 a) Simulating pairs of policies b) maxmin player chooses move i which leads to the maximum value c) minmax player chooses the best counter move i 21
3.3 On the left, a simple payoff matrix for the game of Rock-Paper-Scissors. On the right, a sketch of the payoff matrix used in the RTS game simulations. S_i represents policy i. 22
3.4 Updated game cycle 24

4.1 Simulator main loop pseudo-code 30
4.2 The simulation timeline 31
4.3 Pseudo-code for part of calcBestPolicy() function. 32
4.4 Snapshots of a typical map and the progression of a game. Light gray is a static player playing the Spread Attack policy, black is the RTSplan player. 34
4.5 Function for determining the next time of interest. 37

Chapter 1

Introduction

1.1 Real-time Strategy Games

A popular genre of computer games on the market today is real-time strategy (RTS) games. In a typical "deathmatch" RTS game, players gather resources and build structures and units with the ultimate goal of using those units to destroy the units and structures of the enemy. Other game types include Capture the Flag (CTF), Siege (no attacks are allowed for a specified starting time period), Conquest (capture of strategic points), Team Play, and many others. Most RTS game types involve starting out with limited resources, a few workers, and a town centre. Players begin by collecting resources, developing their base using the acquired resources, and building a military force. Use of resources is commonly divided between building defensive structures to protect the base from enemy attacks, building units to scout the surrounding areas for additional resources and the locations of enemy bases, creating a military force for defense or offense, researching technology, and creating structures that allow the creation of more advanced units. Examples of popular commercial RTS games are Starcraft [3], Age of Empires [12] and Red Alert [38].

In several respects RTS games are different from classic games such as Chess, Checkers and Go. They feature dozens of unit types, several types of resources and buildings, and potentially hundreds of controllable units. Furthermore, unlike most classic games, all players make their moves simultaneously and in real-time. In general, RTS games are fast-paced; any delay in decision-making can lead to defeat. Adding to these difficulties is a high degree of uncertainty caused by restricted

Figure 1.1: A typical Starcraft situation, where a base is attacked by enemy forces.

player vision which is usually limited only to areas within the sight range of allied units and buildings (see Figure 1.1).

1.2 RTS Game Strategic Areas

Playing RTS games well requires skill in the following areas:

1. **Resource and Town Management.** Decisions must be made about how many resources to collect, where to search for them, and when to look for additional resources. Players must also decide when and where to build which structures and when to train which units. Furthermore, in most commercial RTS games, players have the option to upgrade their units' capabilities at the cost of time and resources.

2. **Combat Tactics.** When opposing armies meet, individual units must be given orders on who to attack, where to move, and which special ability to execute. Better execution of special skills and focusing of firepower can lead an army with material inferiority to still achieve victory.

3. **Army Deployment.** Once a player has built several units, these units need to be assigned to groups and be issued orders on what to do, e.g. defend a base, attack enemy encampment, and/or move to a location.

Each one of these areas can be treated as a separate planning task, with its own level of abstraction. Humans do a good job of dealing with these tasks, either separately, or in combination. The challenge of artificial intelligence (AI) systems in this domain is to do as well, or better than their human counterparts. They also need to address a few other lower level planning issues, such as pathfinding for example.

1.3 Commercial RTS Game AI

AI systems in today's commercial RTS games are for the most part scripted [29], with the common exception being pathfinding, which usually uses A* or a similar algorithm. For example, the AI player usually follows a precise set of instructions at the start of the game to develop its base. Once this script achieves its goal condition, the AI system will switch over to a new sequence of instructions (script), and start to follow them, etc. While this method gives the AI the ability to provide a challenge to a human player, it does have several limitations.

First, the AI has a limited set of scripts, and thus its behaviour can quickly become predictable. Second, because every script needs to be created by human experts and takes time to implement and test, developing a strong scripted AI system for an RTS game can become a major undertaking. Finally, scripts can be inflexible and any situation not foreseen by the script creators can lead to inferior game play.

To compensate for these shortcomings, current commercial RTS game AI systems are given extra advantages, usually in the form of more resources, extra knowledge of the game state, or sometimes even by allowing the system to "cheat".

While this approach is acceptable in campaign modes that teach human players the basic game mechanics, and to provide a challenge for casual players, it does not represent a solution to the RTS game AI problem of creating systems that play at an expert human level in a fair setting.

1.4 Computationally Difficult Properties of RTS Games

There are several reasons why currently no strong RTS game AI players exist:

1. **Complex Unit Types and Actions.** Unlike Chess, which has only 6 unit types, RTS games can have dozens of unit types, each with several unique abilities. Furthermore, units in RTS games have several attributes such as hit points, move speed, attack power, and range. In contrast, Chess units each only have one attribute: their move ability. This leads to a much larger state space, and thus the performance of traditional search techniques such as alpha-beta will suffer.

2. **Real-Time Constraint.** Tactical decisions in RTS games must be made quickly. Any delay could render a decision meaningless because the world state may have changed in the meantime. This real-time constraint complicates action planning further because planning and action execution need to be interleaved.

3. **Large Game Maps and Number of Units.** Maps in RTS games are larger than any classical game board. Checkers has 32 possible positions for pieces, Chess has 64, Go has 361. By contrast, even if the RTS game does not happen to be in a continuous space, there are hundreds to thousands of possible positions a unit could occupy. Thus, reaching a goal could involve hundreds of individual steps instead of the usual ten or twenty, thereby greatly increasing the search depth of a traditional algorithm. Furthermore, the number of units in an RTS game can reach hundreds. Because the branching factor grows exponentially with the number of units, applying traditional search techniques quickly becomes infeasible.

4. **Simultaneous Moves.** Units in RTS games can act simultaneously. This presents a problem for traditional search techniques, because the action space becomes exponentially larger.

5. **Several Opponents and Allies.** Typical RTS game scenarios feature more than one opponent and/or ally. This presents yet another challenge to traditional AI techniques. Though some work exists on AI for turn-based n-player games [31], there are currently no solutions able to run well in real-time.

6. **Imperfect Information.** RTS games are played with imperfect information. Enemy base and unit locations are initially unknown, and decisions must be made without this knowledge until scouting units reveal it. Furthermore, because moves are simultaneous, the next micro-move made by the opponents' units is always unknown. Currently, there are no AI systems that can deal with the general incomplete information problem in the RTS game domain. However, some recent work on inferring agent motion patterns from partial trajectory observations has been presented [30]. There have also been some promising results obtained for the classic imperfect information domains of bridge [16] and poker [2] which may be applicable to RTS game AI.

Due to these properties, creating a strong AI system for playing RTS games is difficult. A promising approach is to implement a set of expert modules for sub-problems such as efficient resource gathering, scouting, and effective targeting, and then to combine them. For example, there could be a module that solely deals with scouting the map. The information gathered by the scouting expert could then be used by an army deployment AI, or the resource manager AI. The advantage of this approach is that the complexity of each sub-problem that each individual expert deals with is much smaller than if the problem was dealt with all at once. This thesis will concentrate on the sub-problem of army deployment. The system for this task will not have to worry about resource gathering, building, scouting, or even small-scale combat. Instead, it will make decisions on a grander scale, like how to split up forces and where to send them. This work builds on ideas presented in [7], where Monte Carlo simulation was used to estimate the merit of simple parameterized plans in a CTF game. Here, we approach this problem slightly differently, by combining high-level policy (strategy) simulation with ideas from game theory.

1.5 Thesis Contributions

The contributions of this thesis are:

1. Design and implementation of a simplified RTS game engine that operates on an abstract model of common RTS games.

2. Design and implementation of a simulation-based planning framework used for planning in domains with stochasticity, simultaneous moves, multiple unit types and real-time constraints.

3. Introduction of "fast-forwarding", a concept that greatly reduces the computational overhead of simulations. This concept has been implemented in the aforementioned RTS game engine.

4. Design, implementation, and analysis of the performance of the RTSplan algorithm, which is used for real-time decision making in simplified RTS game scenarios.

5. Design, implementation, and analysis of the performance of an opponent modelling extension to RTSplan.

6. Characterization of parameters affecting the performance and execution time of RTSplan and RTSplan with opponent modelling.

1.6 Thesis Organization

An overview of related research and current state-of-the-art in AI for RTS games is discussed in Chapter 2. The RTSplan planning algorithm and its opponent modelling extension is discussed in Chapter 3. Experimental results, along with details of the implementation of RTSplan in an abstract RTS game world simulator are presented in Chapter 4. Conclusions and discussion of future work is presented in Chapter 5.

Chapter 2

Planning in RTS Games

This chapter is an overview of the various planning issues that are routinely part of RTS games. We describe the current state-of-the-art algorithms for various RTS game AI issues and discuss simulation-based planning (SBP) [21], which is an alternative method for planning that can be used for abstract planning purposes in RTS games.

2.1 Planning Overview

Classical planning problems have traditionally been single agent problems. They usually deal with finding an action sequence that will take an agent from its start state to a designated end state. A common example of classical planning is pathfinding, which focuses on providing a path, often the shortest path, from one location to another.

Classical planners make the assumption that if there are other agents in the world, they are working cooperatively with them in trying to achieve the same goals. Little provision is made for having to deal with an agent who is trying to prevent them from accomplishing their goals, while trying to achieve its own separate goals at the same time.

Adversarial planning deals precisely with this issue. Most adversarial planning tasks occur in the area of games, such as Chess, Go, Poker, etc. However, it is used in other planning areas as well. For example, it can be used in situations where non-determinism is caused by known, but uncontrollable actions of the environ-

ment [17].

A widely used adversarial planning algorithm for turn-based games in discrete space with perfect information, is the minimax algorithm (and its alpha-beta enhancement). It has been used successfully in Chess and Checkers, producing superhuman strength programs [28].

However, unlike Chess and Checkers, minimax has not been used successfully in the game of Go, mainly due to the large search space presented by the game. There has been some progress made in this area through the use of HTNs [39], which has had some limited success. In this implementation, two opposing agents are assumed to be competing by trying to complete their goals while preventing the opponent from completing theirs. A search of the interacting plans of the agents is then performed, and if a goal state is reached by either agent, backtracking is performed and the next branch of the plan search tree is examined. Although this approach was fairly successful in solving many beginner-level life and death problems in Go, it and minimax do not translate well into RTS games for several reasons:

- RTS games feature simultaneous moves, while the above-mentioned approaches assume an environment where the players alternate moves.
- Adversarial planners assume a perfect information environment, which is rarely the case in RTS games.
- RTS games require decisions to be made on the fly, and thus there is no time to stop and wait while expensive planning is performed.

Real-time planning is concerned with the last point on the preceeding list. A common application of real-time planning is real-time scheduling. A process scheduler in a computer's operating system is an example of a common real-time planning algorithm. However, RTS games require adversarial real-time planning, and that is a more difficult problem due to the increased complexity of dealing with a hostile agent.

2.2 Planning Methods Relevant to RTS Games

Classical planners are not designed to deal with imperfect information, real-time constraints and simultaneous moves. However, there are several newer planning methods which have been used, or have potential to be used in the domain of RTS games. Some of these are described below.

2.2.1 Hierarchical Task Networks

Classical planning frameworks such as STRIPS [14] have been around for a long time and have matured significantly over the past decades. However, classical planning algorithms make the assumption that the world remains static during the planning and execution of plans. This assumption does not hold in RTS games, where the world state is constantly changing, and there is no opportunity to "pause" while an expensive planning calculation is performed.

The lack of suitability of classical planners has resulted in most traditional RTS game agents to be mainly reactive. Many of the agent's responses are hard-coded by the developer, and perform no planning at all. These hard-coded responses tend to be simple, such as a worker unit automatically running away if attacked, or a soldier returning fire when an enemy comes within range. Although this approach has the advantage of being computationally inexpensive, it has several problematic issues. First, it puts a burden on the developers, as they have to implement a response for every possible situation that can occur during a game. This becomes more problematic as the RTS game genre evolves and worlds become more complex. Furthermore, this approach makes it difficult to achieve any degree of reasonable cooperation between the various reactive agents because they have little knowledge of the motives and orders of other agents [37].

To adapt classical planning algorithms for real-time games, game developers have begun to use an approach that abstracts various planning tasks, and creates a hierarchy of tasks. This approach is called a hierarchical task network (HTN). Unlike computationally expensive algorithms that use limited abstraction, or highly abstract plans which have trouble controlling the individual units in an RTS game,

HTN planners create several levels of abstraction, with each level having its own separate plan. For example, an "attack base" task at the highest level could be decomposed into two subtasks, "gather army", and "move army to enemy base". Each of these subtasks could have its own subtasks, with a different set of operators for each one [37]. Examples of HTN planning systems include UMCP [13], SHOP2 [25] (used for playing brige) and O-Plan [8].

For dynamic environments, HTN planners are more effective than classical planners (best case is linear complexity, instead of exponential when planning in dynamic worlds [37]) performance-wise. Furthermore, because plans are composed of several different subtasks, partial replanning is possible because only a few subtasks may be causing the invalidation of a plan, and thus only they would need to be replanned. HTN planners also allow for some degree of cooperation between agents or groups, if the right abstraction level is used. This is usually accomplished through the creation of special actions which allow the different agents or groups to synchronize with each other, for example, if two different groups want to attack an enemy base at the same time from different directions.

HTNs are becoming more common in commercial real-time games recently. For example, the game Full Spectrum Command uses a task system which has two types of tasks [11]: composite tasks, which can be composed of other composite tasks or simple tasks, and simple tasks, which are composed only of actual actions given to the agent. This allows for an arbitrary hierarchy of tasks to be created.

2.2.2 Monte Carlo Approach

One promising method for dealing with difficult or complex domains such as RTS games is Monte Carlo sampling. This generally involves playing a large number of randomized games from the current position, and then evaluating which moves lead to the highest score on average. One classic example of where this has been successful is with the introduction of rollouts (Monte Carlo sampling runs) in backgammon [40] (e.g.,TD-Gammon [32]).

Monte Carlo Go

The Monte Carlo sampling approach has been successfully used in recent computer Go programs such as OLGA [5] and MoGo [15]. In these programs, moves from a given position are evaluated based on how well they performed over a large number of random games played from that position and beginning with the move being evaluated. Moves are generally selected at random from all available moves, although obviously inferior moves are not included. This process continues until the game is played out in its entirety, and then the moves are scored. The evaluation of the move at the starting position is the mean value of all the scores of games played with that starting move.

The Monte Carlo approach is highly dependent on processing power, because having a faster processor means more games can be played out, leading to more accurate move evaluations. Furthermore, Monte Carlo sampling also lends itself nicely to parallelization, because it is easy to offload different random games to multiple processors and to recombine the results. This is becoming a more significant advantage with the introduction of multi-core architectures.

MCPlan

Although Monte Carlo sampling has been used with success in turn-based games such as Go, it still remains to be seen if the same approach can be applied to RTS games. Recently, Monte Carlo planning was applied to RTS games by Chung et al. using their MCPlan algorithm [7]. MCPlan uses the same ideas which are sometimes used in Go with Monte Carlo sampling. To apply it to the RTS game domain, low level unit actions are abstracted into plans.

Abstraction is necessary in the domain of RTS games, as working with atomic moves, such as “move left” or “move up”, is infeasible due to the large state space present in RTS game maps. Essentially, instead of generating random moves like in Go, random plans composed of low level abstract actions such as “attack base” or “explore” are formed and then simulated versus the randomly generated plans of the opponent [7].

Furthermore, unlike in OLGA and MoGo, MCPlan does not randomly simulate

until the end of the game. Instead, it simulates to a specified maximum length of action sequences, and then uses an evaluation function to judge the game state. This is possible because, unlike in Go where creating a good evaluation function is difficult, the value of a state in RTS games is significantly influenced by the material difference.

MCPlan has shown some promising results, although its performance has yet to be tested against highly-tuned scripted AIs or humans [7].

2.2.3 Simulation-Based Planning

The basic idea of Simulation-Based Planning (SBP) [20] is to take a series of given plans, use a simulator to execute each one of them, and then evaluate the result of using that plan. Based on these results, the agent can then choose the plan that leads to the best outcome. If the domain has uncertainty or randomness, plan simulations can be repeated to get more statistically significant results [19]. It has successfully been used in route planning, controlling a truck depot [19] and even some military planning [22]. This suggests that it lends itself well to adversarial environments and environments with uncertainty and randomness. This also means that it could be applied to RTS game planning as well.

However, one of the drawbacks of SBP is that it underestimates the actions of the opponent. Adversaries are assumed to be simple for efficiency purposes when run in the simulation [20], and this does not properly represent what the "real" opponent is likely to do. For example, Lee describes an air interdiction scenario where an allied airplane must avoid enemy SAMs and fighters while attempting to reach a specified bomb target [22]. However, no real intelligence is given to the opponents. Their ability to move/detect is represented simply by creating a radius based on the object speed and/or detection area around each object (plane or SAM) and assuming that they project their influence within that radius. While this approach works for short-term scenarios such as going from point A to point B while avoiding certain areas of the map, it does not work in longer term situations. With this assumption of a simplistic opponent, any long-term scenario would effectively yield a radius that covered the entire operational map because any mobile opponent could have moved

to any location in the long term. This suggests that SBP could be used successfully in RTS games for situations that require short-term planning, such as assaulting an enemy base while avoiding immobile defensive towers, but would likely struggle in planning a long-term campaign against a highly mobile opponent.

Furthermore, without a proper abstract model of the world, SBP is computationally expensive. Planning simulations must run *significantly* faster than the actual real world simulation to be used in a real-time setting, otherwise data used in the SBP will be out of date. Thus, for SBP to work in RTS games, a well designed abstract model is needed, as is the case in our approach as well.

Overall, SBP fits fairly well into the RTS game planning domain. Like our approach, it uses an abstract view of the world, combined with forward simulations in its decision making. However, the lack of an advanced opponent model in SBP suggests that successful planning is limited to short-term objectives. Conversely, due to the simplistic opponents, SBP is likely to perform faster than our approach. Thus, perhaps an ideal approach would use a combination of the two, with SBP dealing with short-term planning and planning against stationary enemies, while our approach being used for the long-term situation. This combination is not explored in this thesis, but could be the subject of future work.

2.3 AI in RTS Games

Planning is already being used in domains similar to RTS games. For example, it has been used in Robocup [1], a popular robotic soccer challenge, and F.E.A.R [26], a first person shooter game. Existing RTS games currently do not use high-level planning, however, they employ the following techniques to aid their AI.

2.3.1 Map Analysis

Terrain analysis is a useful tool in RTS game AI. It provides the AI with an abstract view of the world which aids it in decision making. Multiple systems in an RTS game will make use of this information, whether it be for deciding where to build a base, or where the enemy forces are most concentrated, or any other matter. There

are different methods of terrain analysis, two of which are described below.

Analysis Through Pathfinding

Some terrain analysis is accomplished simply through using the game's pathfinding system [9]. For example, the pathfinder could be used for determining which parts of the terrain are currently reachable, or determining the desirability of resource patches based on the distance they are from the player's base. This approach has the advantage of being highly accurate, however it is computationally expensive [9].

Influence Maps

Influence maps are commonly used in RTS games as a way to perform spatial reasoning. They are essentially two dimensional arrays which are an abstraction of the world map. Each entry in the 2D array corresponds to a specific section of the map, and stores a numerical value. The numerical value can indicate many things, for example the strength and location of enemy or friendly forces, location of resources for gathering, potential chokepoints, etc. [33].

The reason this technique is called influence mapping is because each cell influences its surrounding cells. The influence decreases as the distance from the cell increases, usually with some sort of falloff rule. Cells exert an influence to better reflect that enemy units are usually not stationary, and thus could move to a nearby location. Due to this, the falloff rule depends at least partially on the speed of the unit.

An extension of the basic influence map, spatial databases are essentially a collection of influence maps layered over each other on the world map [35]. For example, one layer could consist of an influence map that keeps track of only enemy units, while another layer could keep track of areas of the map that have not been explored recently. These layers can then be queried by various AI systems to help with reasoning. For example, the scouting AI could query the exploration influence map to see where to explore next, while the attack AI would query the enemy strength map to determine if attacking at a particular location would be a good idea. Multiple layers are also combined to create a "desirability" layer, which can then

be used for various other reasoning tasks. For example, the expansion AI will combine the enemy location map with the resource map to make sure it does not try to expand to an area that has a strong hostile presence.

2.3.2 Dependency Graphs

Dependency graphs are a useful aid for strategic AI in RTS games. Essentially, they are graphs which represent the various dependencies in an RTS game. For example, prior to building an infantry unit we may be required to build a barracks, while a barracks is dependent on having sufficient resources and having a town center [34].

Dependency graphs allow the AI to make various inferences about the game. For example, if a scout spots an airport at an enemy base, it can then infer that the enemy also likely has air units. Conversely, if we see air units attacking us, we know that the enemy has at least one airport.

Identifying weak nodes in a dependency graph is also useful for planning strategic actions. For example, if the enemy is known to have several units and plenty of resources but not many farms (which are required for creating more units), the farms would show up as a weak node in the graph. This is because they have many units dependent on them, and thus the AI may decide to focus on destroying the farms.

2.3.3 Multi-Tiered AI

Designing effective algorithms in RTS games that form cooperative or complicated strategies while considering individual units is difficult. To develop large-scale and complex plans, the complexity that comes with considering large numbers of individual units must be reduced. An excellent way of reducing this complexity is to collect individual units into squads [18]. For example, by grouping units into squads of eight, the complexity can be reduced considerably. Due to the reduced complexity, and thus reduced burden on the developers, squad tactics have recently become more common in many RTS games [18].

The idea of squads can be further expanded to include the creation of platoons, companies, armies, and so on. In fact, to deal with the large complexity of thou-

sands of individuals, real world armies have been using this hierarchical approach for thousands of years [18].

One of the main advantages of having a tiered AI system is that each tier only has to concern itself with information relevant to its decision making. The highest level commander AI does not deal with pathfinding or formation issues, while similarly the lowest level AI for a soldier does not care about the current position of the enemy army. Tiers can also communicate with each other, with the lower-level tiers generally providing information to the higher-level tiers, while the higher-level tiers pass down orders to the lower-levels [18].

Several different AIs can exist on the same tier as well. For example, separate high tier AI systems could exist for scouting, resource gathering, war waging, base building, etc. In general, tiered AI systems make the creation of complex strategies much easier on the developers, and thus they are becoming more commonplace in strategic games today.

2.4 Conclusions

The RTS game domain is difficult from an AI perspective. It features imperfect information, simultaneous moves, adversaries, hundreds of complex units, dozens of units types, and real-time constraints. Any one of these properties is problematic to deal with when trying to create an AI system. Thus, it is not surprising that there is a distinct lack of high-performance AI solutions in the RTS game domain.

Currently, the state-of-the-art AI in RTS games is hand-coded or scripted rules for that particular game. While this approach can yield challenging computer opponents, it can make the AI appear repetitive, or too predictable to provide a challenge to an experienced human player; scripts can only go so far. Developers can end up giving the AI unfair advantages, such as extra resources or full world knowledge to make the AI more competitive.

Unfortunately, classical planning techniques do not scale well to all of the various issues inherent in the RTS game domain. A powerful approach would be to take a given set of scripted policies, and choose the most appropriate policy based on the

strategic situation at the time. This can be achieved through forward simulation of these policies, and is the main focus of this thesis.

Chapter 3

Simulation-Based Adversarial Planning

Creating a search or simulation-based planning algorithm for RTS games is difficult. Much of this difficulty stems from the large complexity of the RTS game domain and the real-time constraints present in RTS games. Traditional state space search techniques cannot deal with RTS game domains. This has led commercial AI game developers to invest time and effort into producing script-based computer players which are essentially just complex finite state machines.

Our approach takes the idea of using scripted policies which, as discussed earlier, can be effective in RTS games, and add a layer of planning on top. Essentially, we try to improve the performance of a computer controlled player by adding a mechanism (RTSplan) for choosing a script (policy) from a given set of available scripts. This is achieved by determining the merit of each script. The merit of a script is determined by simulating it some time into the future and evaluating the result of using that script. Then the script that is most likely to yield the best result is chosen as the one to execute until we perform another planning cycle.

3.1 Action Abstraction

Abstractions are required before state-space search algorithms can be applied to complex decision spaces such the ones faced in RTS games. Not only is the domain complex, with micro-moves and many units, but our approach also performs multiple lookaheads (simulations), and is under real-time constraints as well. This

means that each simulation must run *significantly* faster than "real" world time, because several need to be performed before planning is complete. A natural way to increase the speed of simulations is to reduce the complexity of the domain by means of abstraction, and that is the approach we take.

Some types of abstractions are commonly used today. For example, the use of spatial abstractions can speed up pathfinding considerably while still producing high-quality solutions [4]. Likewise, temporal abstractions, such as time discretizations, can help further reduce the search effort.

Here, we explore the abstraction mechanism of replacing a potentially large set of low-level action options by a smaller set of high-level policies from which the AI can choose. Policies are considered decision modules, functions of state to action. Policies can range from complex maneuvers that involve all of a player's units (such as "use all units to attack the least defended base while avoiding combat"), to simple commands (such as "move to position X"). Policies can even be created by randomization, such as creating a policy which is composed of two consecutive move commands to random locations.

Consider the various ways of playing RTS games. One typical policy is "rushing", where a player produces a small fighting force as quickly as possible to surprise the opponent. Another example of a typical policy is "turtling", in which players create a large force at their home base and wait for others to attack.

It is relatively easy to implement such policies which, for the purpose of high-level planning, can be considered black-boxes, i.e., components whose specific implementation is irrelevant to the planning process. We implement policies as scripts, with a script being a sequence of actions, triggered by events, not unlike scripts used in RTS games today.

3.2 Planning Based on Policy Simulation

The task of the high-level planner then is to choose a policy to follow until the next decision point is reached, at which point the strategic choice is reconsidered.

The aim of this scheme is to create a system that can rapidly adapt to state

changes and is able exploit opponents' mistakes in highly complex adversarial decision domains.

Game Cycle

```
while (game not finished){
  if (planning finished)
    chosen policy set as current policy
    begin planning next policy
  else
    // planning is time constrained, and will
    // return even if planning is not finished
    continue planning

  continue executing current policy
}
```

Figure 3.1: Top level planning cycle

The cycle shown in Figure 3.1 allows planning and execution to be interleaved. While we are planning what our next policy should be, we continue to execute our current policy. Thus, the planning process can run in real-time because we plan and execute at the same time. It also shows that we continuously replan, trying to determine the best policy for the current world state which is continually changing.

3.2.1 The RTSplan Algorithm

With the game cycle established, the question now becomes: Having written or been provided a number of policies, how do we pick the appropriate one in a given situation? Assuming we have written or have been provided with the set of policies the opponent can choose from, we can learn about the merit of our policies by simulating policy pairs, i.e. pitting our policy i against their policy j for all pairs (i, j) and storing the result in a payoff matrix R, at location r_{ij} (Figure 3.2a).

Because planning and execution are interleaved, with computation spread over several cycles, payoff matrix R is filled entry by entry. One or more entries in R are computed, based on available CPU time, each cycle, and the new policy is not chosen until the entire matrix has been fully populated. Once the matrix

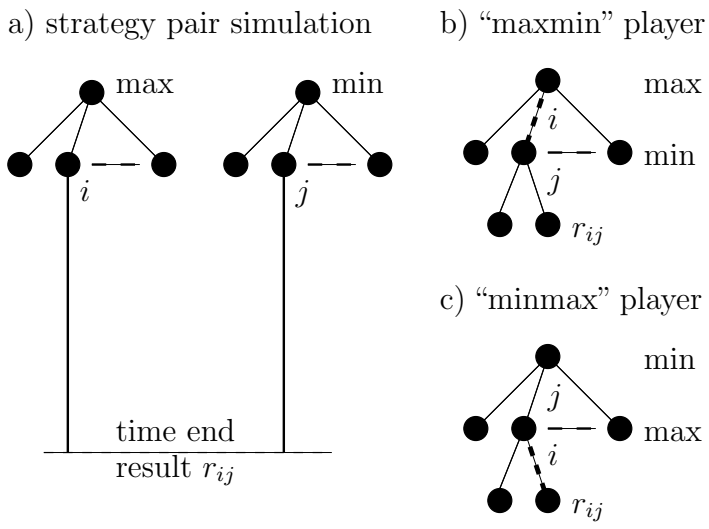


Figure 3.2: a) Simulating pairs of policies b) maxmin player chooses move i which leads to the maximum value c) minmax player chooses the best counter move i

is populated, the appropriate policy can be chosen in several ways. We can use minmax, or maxmin, etc. or alternatively, a linear program (LP) to choose the best policy.

In the minimax rules, one player (max) maximizes its payoff value while the other player (min) tries to minimize max's payoff. The two variants with either player max or min to play first are depicted in Figure 3.2 b) and c). In these examples, player max plays move i, which leads to the best minimax value. Only in case where there are pure Nash equilibrium policies do the payoffs coincide.

In a zero-sum two-player setting with simultaneous moves, the natural move-selection choice then would be to determine a Nash equilibrium policy [24] by mapping the payoff matrix R into a linear programming problem whose solution is a probability distribution over our policies. In the Nash equilibrium case, neither player has an incentive to deviate. Nash-optimal policies can be mixed, i.e. for optimal results, policies have to be randomized — a fact which is nicely illustrated by the popular Rock-Paper-Scissors (RPS) game.

In RPS, players select a move simultaneously between three possible moves: Rock, Paper, or Scissors. Scissors wins versus Paper, Rock wins versus Scissors, and Paper wins versus Rock. The payoff matrix for Player A in a game

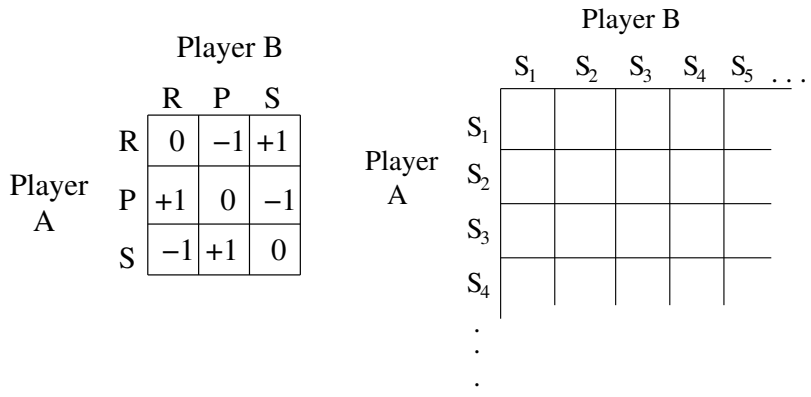


Figure 3.3: On the left, a simple payoff matrix for the game of Rock-Paper-Scissors. On the right, a sketch of the payoff matrix used in the RTS game simulations. S_i represents policy i.

of RPS is shown in Figure 3.3. The Nash-optimal strategy is to choose each action uniformly at random; in particular $P(\text{Choose Rock}) = P(\text{Choose Paper}) = P(\text{Choose Scissors}) = \frac{1}{3}$.

In the case of our RTS game planner, the actions are instead policies and the payoff values are obtained via results of simulations into the future. In the simplest version, policies are simulated to completion or until they time-out, in which case a heuristic evaluation function is necessary to estimate who is ahead.

Informing the opponent about the move choice can be detrimental, like in Rock-Paper-Scissors, and the Nash-optimal strategy may have advantages over maxmin, or minmax, or both. For example, if we revealed to our opponent that our policy was to attack all their bases at once, they could respond with an appropriate counter policy, such as massing their army and killing our attack forces off one by one. This is the reason why we model RTSplan as a simultaneous move game instead of an alternating move game.

The following describes the simulation approach to selecting an appropriate policy from a given set of policies (RTSplan Algorithm):

- Consider a set of policies P of size n and compute each entry of the payoff matrix R (with dimensions of $n \times n$) by assigning policy p_i to the simulation-based AI, and policy p_j to its opponent, and executing these policies until

either there is a winner, or a timeout condition is reached. Once the simulation is completed, the terminal game value or a heuristic evaluation is assigned to payoff matrix entry r_{ij}.

- Calculate a Nash-optimal policy with respect to our player using the standard Linear Programming (LP) based method, with the LP formulation given below, or alternatively a minmax or maxmin move. The same method was used to solve the Oshi-Zumo game [6].

- In case of the Nash-optimal player, assign a policy randomly to our player, using the probability distribution returned by the LP solver. For the minmax or maxmin players, play the minmax or maxmin move directly.

- Repeat from step 1 as often as is desired while executing the chosen policies.

Step 2 in the RTSplan algorithm requires the solving of a LP in order to calculate the Nash-optimal policy for the given payoff matrix R. This formulation is also valid for the opponent modelling extension discussed in the next section, and implements von Neuman's Minimax theorem [36]: $\max_x \min_y x'Ry = \min_y \max_x x'Ry = Z$, which states that there exist mixed equilibrium strategies x and y with maximum payoff Z for player 1 and $-Z$ for player 2.

An LP characterizing such Nash-optimal mixed policies for the maximizing player is the following:

$$\begin{aligned}
&\text{Maximize Z such that} \\
&\text{for all } 1 \leq j \leq m : Z \leq \sum_{i=1}^{n} r_{i,j} x_i, \\
&\text{for all } 1 \leq i \leq n : x_i \geq 0, \sum_{i=1}^{n} x_i = 1
\end{aligned}$$

In our application, $r_{i,j}$ is the simulation result when we choose policy i and the opponent chooses policy j. x_i is the probability of respective policy i for our player. Z is the expected payoff for our player. Our player (max) has policies $1, ..., n$ to choose from, while the opposing player (min) has policies $1, ..., m$. In this case, $n = m$ since our opponent uses the same policies as our player. This is not the case in the opponent modelling extension presented in the next section. To solve this LP, we used an LP solver based on the implementation in [27].

3.3 Opponent Modelling

One problem with the Nash equilibrium strategy is that it does not exploit our opponent's mistakes. For example, in Rock-Paper-Scissors, the Nash-optimal play is to randomly select one of the three moves with equal probability. Although this guarantees that our opponent cannot exploit our play, it also means that we cannot exploit theirs. For example, a Nash-optimal player would only be able to break even against a player that played Rock all the time; an exploitative player would learn this behaviour and exploit it in order to win.

To exploit an opponent's policy, we first require some idea of what their policy may be (a model of our opponent). Once we have an idea of what our opponent is doing, we can respond with an appropriate counter-policy. In our case, opponent modelling involves observing the opponent for a specified period of time, followed by choosing a subset of policies from our player's policy set that the opponent appears to be executing. Then the next step is to find the best-response policy to this subset. Opponent modelling requires changing the game cycle, and the updated game cycle can be seen in Figure 3.4.

```
while (game not finished){
  if (planning finished)
    chosen policy set as current policy

    // update opponent's subset periodically
    if (sufficient time has passed)
      update opponent's policy subset

    begin planning next policy
  else
    // planning is time constrained, and will
    // return even if planning is not finished
    continue planning

  continue executing current policy
}
```

Figure 3.4: Updated game cycle

Parameters

There are three parameters that will affect the performance of opponent modelling:

1. **Prediction cutoff ratio (f):** A real number in [0,1] used to calculate the cutoff point for a policy's comparison value to determine whether it will be added to P'.

2. **Time Period (t):** The interval in game seconds between recalculation of active policy set P'.

3. **Distance Measure (D):** Measure used for calculating the difference between two states. We discuss the specific implementation for our domain in greater detail later in this section.

These parameters are domain dependent, and thus should be determined after some experimentation has been performed on the domain to which this algorithm is applied.

While the original payoff matrix R has dimensions of $n \times n$, the matrix of the opponent modelling extension (R') has dimensions $n \times m$, where m is the size of the active policy set P' of the opponent.

The active policy set P' is composed of policies from policy set P. A policy p is considered *active* if the current world state resembles a state that would be reached from a state time period t earlier, with our opponent using policy p for that specified time period t. Essentially, if we run a simulation over the past time period t with the assumption that our opponent is executing policy p, and it turns out that the reality is similar to our assumption, we add p to P'.

This approach works well at detecting if the opponent is executing a policy in our policy set and for policies similar to that policy as well. It can also adapt if the opponent changes policies because P' is recalculated periodically. However, it does not keep a history of previous policies, nor does it keep a table of weights, or an artificial neural network, to keep track of the opponent's previous tendencies, as is done sometimes in Poker [10]. The effectiveness of these alternative methods of opponent modelling have not been explored in this thesis.

The active policy set P' is determined by the following process:

The RTSplan-O Algorithm

- Save the current game state as state s_a. This includes storing the positions of all the objects in the game world, the current policies employed by both players, and any other relevant information.
- After time period t has elapsed, the current game state is once again saved, this time as state s_b. Let A be the policy that was used by our player over time period t.
- For each policy in P, simulate for time period t from starting state s_a, with our policy A and policy p_i for the opponent to yield state s_i.
- For each state s_i, compare s_i to s_b using a distance measure D between the two states and store the result as d_i. Let d be the average over all of d_i. Let $c = d \cdot f$, where f is a specified ratio value. In essence, c represents the range that the difference between two states can be within in order to be considered "active".
- For each state s_i, if $d_i \leq c$ then p_i is added to the active policy set P'.
- If the previous steps resulted in $|P'| = 0$ then set $P' = P$.

The new payoff matrix R' for the RTSplan algorithm has $|P|$ rows and $|P'|$ columns. Essentially, policies in P' are those which closely match what we expect the opponent to do if they were performing one of the policies in our policy set P. If we cannot determine which policy our opponent is likely executing (their policy is significantly different from anything in P) then we revert back to the pure RTSplan algorithm using the original payoff matrix R.

Distance Measure D

The distance measure used to compare two different states is fairly straightforward, but not without complications. If both states have the same number of objects then

we just tally up the sum of all the distances between corresponding objects in their respective states and return this sum.

Thus, $D(s, s') = \sum_{i=0}^{|s|} \sqrt{(x_i - x'_i)^2 + (y_i - y'_i)^2}$ if $|s| = |s'|$ where (x_i, y_i) and (x'_i, y'_i) represent the positions of objects i in states s and s' respectively.

If there is a difference in the number of objects, meaning that one state had engaged in more combat than the other, and lost or killed units, then we mark that state with a flag. Once all the distance measures are completed, all the D values of the flagged states are set to the maximum non-flagged value. This eliminates the need for an arbitrary "penalty" value given to states with different numbers of objects, thus maintaining a proper average value not skewed by the penalties of the flagged states. If all states were marked as flagged then we skip to the last step of the active policy set determination algorithm.

Complications with this approach can occur if objects cannot be uniquely identified. For example, if the only information we are given about an opponent's army is the number and types of units, two armies with identical compositions would be indistinguishable using the above approach. Furthermore, if objects merge to create less objects, or split apart to create more objects, the method described above would not be sufficient. However, the advantage of our approach is that it is easy to implement and is not computationally expensive.

Our implementation can currently uniquely identify all objects in each state, and does not allow for merging or splitting of objects, and thus does not suffer the problems discussed above. However, creating a general solution that can deal with with the above-mentioned problems should be the study of further work. For example, one possible solution is to create a strategic-based distance measure, instead of the location-based measure we use here. However, this alternative is not explored in this thesis.

Discussion

Opponent modelling has several advantages. First, it reduces the number of simulations needed to be done because $|R'| \leq |R|$. Reducing the number of simulations has a significant impact on performance as the simulations are the most computa-

tionally intensive part of the RTSplan algorithm. Second, assuming the opponent does not deviate from the predicted policies, opponent modelling also leads to better play by RTSplan overall because it does not have to give the "benefit of the doubt" to the opponent by considering all of their available policies in the Nash equilibrium calculation.

However, excluding certain policies from the Nash equilibrium calculations also carries some risk. Including only policies that resemble what the opponent is doing is usually a good idea, but if the opponent is executing a policy that just happens to be similar for time period t does not automatically mean that they are performing the policy we think they are. By concentrating only on how to counter the policy we *think* they are following, it effectively blinds us to other actions they could take, perhaps leading to a loss for us. Also, with our method we are essentially always one step behind our opponent because we are using their past policy as a predictor for their future policy.

Choosing a good value for t and f is particularly important when playing against unpredictable opponents, such as humans. Lower t or higher f values will result in less exclusions, but lead to reduced benefits of opponent modelling. Finding balanced values is essential. Since the effectiveness of these parameters is domain dependent, finding good values will likely require separate experimentation for each different domain. Furthermore, the success of our opponent modelling approach is dependent on the size and variability of our policy set P, with a larger and more variable P leading to better modelling of our opponent because we have more policies for comparison with the opponent's behaviour, and thus a higher chance of obtaining a more accurate opponent model.

Chapter 4

Implementation and Experiments

To test the performance of RTSplan, we implemented a simulator for an abstract model of a typical RTS game. Because RTSplan only considers group management and deployment in the abstract model we use, our experimental setup does not include the handling of resource collection and the creation of units and buildings. This means that we create an abstract model of only a part of a full RTS game. However, its reduced complexity allows us to better concentrate on the army deployment problem. Although RTSplan is designed to work in any abstract model of RTS games, there are some implementation issues specific to the abstraction we use that warrant further discussion.

We conducted several experiments to determine the effectiveness of RTSplan in our abstract model. Similar experiments were performed to test the effectiveness of the opponent modelling extension and to test parameters that affect these algorithms. Finally, the execution times of RTSplan in our abstract model were examined.

4.1 Implementation Details

We have made several assumptions and simplifications when designing our abstract model of RTS games. In particular, combat and pathfinding are simplified to speed up simulations and to concentrate more on the higher-level issues, instead of low-level implementation issues. We also introduce the concept of *fast-forwarding* which greatly speeds up forward simulation.

Simulation Process

```
currTime = 0;

while (!isGameOver()) {

  for (int i=0; i < players.size(); ++i) {
    Policy bestPol = calcBestPolicy(players[i]);
    players[i].updateOrders(bestPol);
  }

  // time increment is based on how much time has
  // elapsed between subsequent calls to this code
  currTime += timeIncrement;

  if(isTimeToUpdateActivePolicies()){
    recalculateActivePolicies();
  }

  updateWorld(currTime);
}

determineWinner();
```

Figure 4.1: Simulator main loop pseudo-code

`calcBestPolicy()` in Figure 4.1 computes the result matrix and returns the best policy, time permitting. If the allowed computation time expires before the matrix is fully computed, it will return null, and the orders for the player will not be updated. For non-simulation based player types, we call the responding hard-coded code block in `calcBestPolicy()`. Regardless of whether a policy was changed in the function, the world advances forward in time by the specified time increment. However, because calculating the best policy may be time consuming, we may have to spread out computations over several world update intervals. This means that the world will continue to advance, even while policy calculations are going on (see Figure 4.2).

For example, in our abstract RTS game, which runs at 8 simulation frames a second when not in real-time mode, the simulator only has 1/8th of a second to

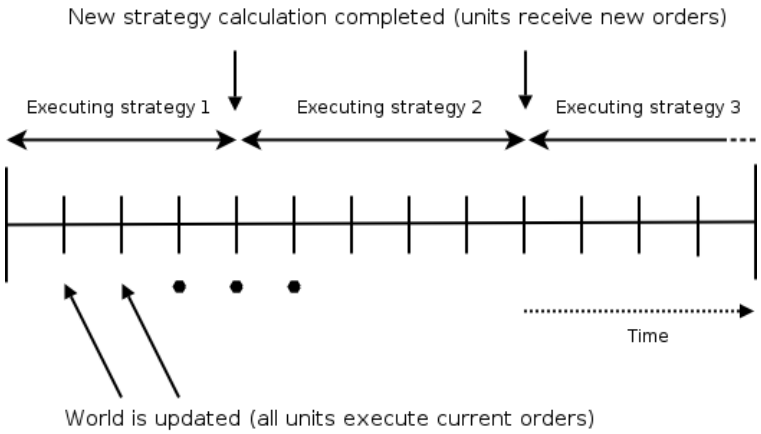


Figure 4.2: The simulation timeline

perform simulation computations before the world advances. This is enough time to compute a few entries of the payoff matrix, but usually not enough time to compute all entries. Thus, all the work done up to that point is saved, and resumed as soon as the real world advances. Once the entire matrix is completed, we can finally determine a policy. It is at this point that actions are being updated. This method allows for the world simulation to run in real-time, because expensive planning is interleaved with world execution. A separate real-time mode exists for non-experimental use, which does not use a constant time increment, but instead uses the length of time elapsed between subsequent calls to the function. We do not use this method in our experiments, since results in that mode are not processor independent.

The method `isTimeToUpdateActivePolicies()` is only used for the opponent modelling extension of RTSplan, and returns true only if the specified time period has elapsed, the *entire* payoff matrix has just been computed, and the new policy calculated. Resizing the payoff matrix before it is completely filled is possible, however, it is not explored in thesis, though observing its effects on performance could be the subject of future work.

Selection of Best Policy

The best policy for our Nash player is calculated in a fairly straightforward manner. First, we need to compute the payoff matrix. Each entry in the matrix represents

the result of a forward simulation between competing policies in which a winner is found or the time limit has been reached. The basic algorithm is shown in Figure 4.3.

```
for (int i=0; i < ourPolicies.size(); ++i) {
 for (int j=0; j < activeOppPolicies.size(); ++j) {

  // this method checks if we have time to compute
  // another forward simulation before real world
  // must be allowed to advance
  if (!nextSimulationAllowed()){
   return null;
  }
  // simulate the competing policies
  r[i][j]=simulate(ourPolicies[i],activeOppPolicies[j]);
 }
}
return pickPolicy(r);
```

Figure 4.3: Pseudo-code for part of calcBestPolicy() function.

There is a check between each simulation to see if there is time to run another simulation without violating time constraints. This can result in the effect that our player is a bit behind the action, because the world is changing while the algorithm is still trying to compute the payoff matrix to determine the next policy. However, for our player to be able to play in a real-time setting, time constraints are necessary, because computing the entire matrix can take too long.

Combat Simulation

RTSplan requires a combat simulator as part of its forward simulation process in order to resolve combat between opposing units within range of each other. Its complexity can be variable, however since RTSplan performs many forward simulations, a very complex combat simulator could adversely affect performance.

In our implementation, we abstract individual units into groups with because we are dealing only with *army* deployment. Not only does this reduce the number of objects that need to be dealt with, but it also more closely matches the way a human thinks when playing an RTS game. A human player usually sends out groups

of units, and deals with individual units only in combat situations. Our combat simulator does not deal with combat tactics. Instead it has a simple combat model that generally favours numerical advantage and is computationally inexpensive.

Pathfinding

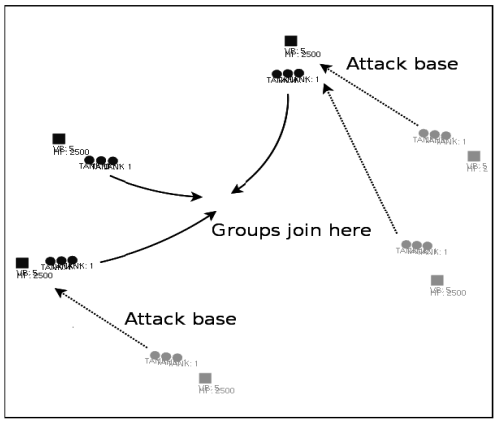
RTSplan requires a pathfinding system that returns a path as a series of waypoints composed of locations. The specific pathfinder implementation is not important, only the format in which the paths are returned.

Our test scenarios did not contain any obstacles, so we did not include any pathfinding system in our implementation. However, RTSplan would work in exactly the same manner if the scenarios had obstacles and we had an appropriate pathfinder.

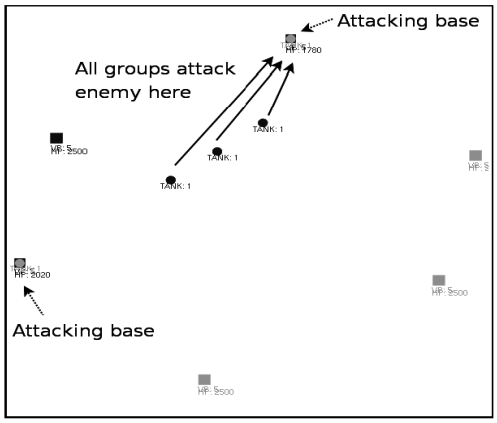
Victory Conditions

In our scenarios there are two sets of victory conditions. One for the actual game, and another for the forward simulations. In our implementation, they are identical. However, this need not be the case, and having a different set of victory conditions for forward simulations from the real game could prove beneficial. This aspect is not explored in this thesis however.

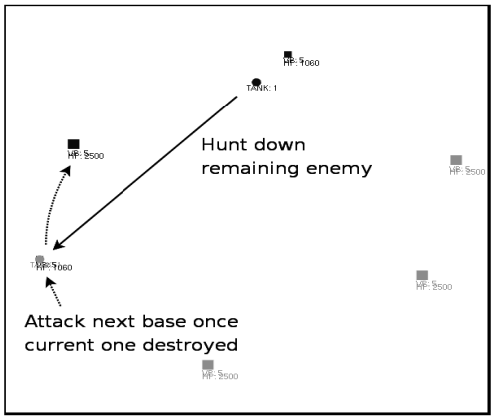
Victory in a scenario is achieved by either destroying all of the enemy's bases *or* units, or by having more bases/units when the game/simulation runs past a predetermined time limit. The time at which to stop the game/simulation is 1000 game seconds, and the method used to break ties is the following: the winner is the one who has more bases. If the number of bases is equal then the winner is the player with the higher number of remaining groups. If either all the bases or groups were killed at the same time, or the material is identical when time runs out, the result of the scenario is declared a tie. The time limit of 1000 seconds was chosen in order to allow the majority of games to play to completion, while still detecting stalemate situations before too much computation time is wasted in simulation. An example of a typical map and scenario progression can be seen in Figure 4.4.



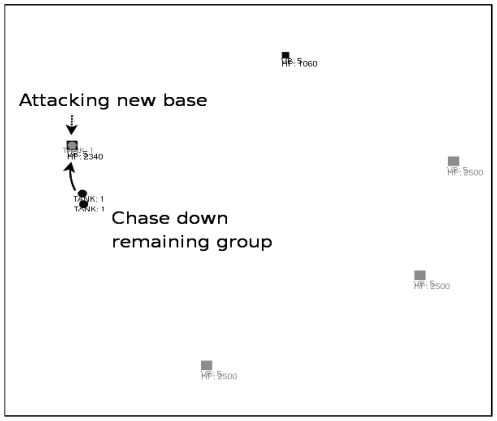
(a) All groups first form local groups of three



(b) Opponent attacks bases while black gathers



(c) Black eliminates part of the enemy force



(d) Black eliminates the rest of the enemy

Figure 4.4: Snapshots of a typical map and the progression of a game. Light gray is a static player playing the Spread Attack policy, black is the RTSplan player.

Policies

All of our initial simulations involved the following 8 policies, chosen mainly based on their diversity and their use by human players in commercial RTS games:

1. **Null**. This is more like a lack of policy. All groups stop what they are doing, and do not move. They do however still attack and defend against any groups or bases within their defined weapon range.

2. **Join up and Defend Nearest Base**. This policy gathers all the groups into one big group, with groups joining at their combined center of mass in order to speed up joining time, and then moves this large group to defend the base that is closest to an enemy group.

3. **Mass Attack**. In this policy, all groups form one large group in the same manner as in the Join policy, which then goes to attack the nearest enemy base until no enemy bases remain. There are two versions of this policy. Given the choice of attacking a base and group within range, one chooses to attack the base first and the other chooses to attack the group first.

4. **Spread Attack**. In this policy, all groups attack the nearest enemy base to them, and this repeats until all enemy bases are destroyed. There are two versions of this policy; the versions are analogous to those of the Mass Attack policy.

5. **Half Base Defense Mass Attack**. This is a split policy. Groups are divided into two halves. One half defends their respective nearest bases, while the other executes the Mass Attack policy.

6. **Hunter**. In this policy, all groups join with their nearest allied group in order to create a larger combined hunting group. After the joining, all of these newly formed groups join into one large group which attacks the nearest enemy group until no enemy groups remain. The initial joining is performed to minimize losses during the secondary join phase.

Policies which require examination of the game state, for example to determine nearest enemy group or base, do so periodically. In our case, the examination is done every 5 game seconds. This periodic examination is due to the fact that we are fast-forwarding via simulation and thus cannot examine the game state continuously. The value of 5 seconds aims to balance the effectiveness of fast-forwarding and the frequency at which policies examine the world state. Fast-forwarding is described in the next section.

4.1.1 Fast-Forwarding of Policies

In our implementation, each forward simulation simulates all the way until the end of the game or to some point in the far future (e.g., 1000 world seconds). Therefore, it is crucial that the simulations of future states are computed quickly. However, these simulations can be expensive, especially if we were to simulate every single time step into the future. To reduce this high cost, we instead calculate the next *time of interest*, and advance directly to this time. This calculated time is derived in such a way that there is no need to simulate any time step in between our start time and the derived time, because nothing interesting will happen during that time interval. Details on what we consider "interesting" are discussed later in this section. The derivation of the *time of interest* is implementation and policy specific, although we expect that many factors in its determination would likely be common among other implementations and policies as well.

Here we introduce the concept of *fast-forwarding*. The idea of fast-forwarding is to advance simulations from one *time of interest* to the next, instead of game tick by game tick, thus greatly reducing simulation cost. The algorithm for finding the next time of interest in our RTS game simulation environment is shown in Figure 4.5.

1. `nextCollideTime()` is calculated by solving a quadratic equation with input being the direction vectors of the two groups in question. The quadratic equation may not be solvable (no collision) or it may produce a time of collision. This is similar to what is used for collision calculations in ORTS [23], another continuous-space RTS game environment. Two groups are considered to be colliding if either one of them is within attack range of the other.

```
double nextTimeOfInterest(){
  // start with maximum value of a double
  double minTime = DOUBLE_MAX;

  // next time opposing groups are in shooting range
  double collideTime = getNextCollideTime();
  if(collideTime < minTime) minTime=collideTime;

  // next time a group's order is completed
  double orderDoneTime = getNextOrderDoneTime();
  if(orderDoneTime < minTime) minTime=orderDoneTime;

  // if units in range, earliest time they can shoot
  double shootingTime = getNextShootingTime();
  if(shootingTime < minTime) minTime=shootingTime;

  // next time policy gets to re-evaluate game state
  double timeoutTime = getNextPolicyTimeoutTime();
  if(timeoutTime < minTime) minTime=timeoutTime;

  return minTime; // time to advance simulation to
}
```

Figure 4.5: Function for determining the next time of interest.

2. `getNextOrderDoneTime()` is a simple calculation. Because all units travel in straight lines from waypoint to waypoint (in our implementation, they travel directly from start to goal due to lack of obstacles), we can just divide the distance to the goal for a group by its maximum velocity. We do this for every group, and return the time at which the first group reaches its goal.

3. `getNextShootingTime()` applies to groups that are already within range of an enemy group and are recharging their weapons. This function returns the next time at which one of these groups can fire again.

4. `getNextPolicyTimeoutTime()` returns the next time that any one of the policies in question is allowed to re-evaluate the game state to give out new orders if necessary.

Fast-forwarding allows forward simulations to safely skip all time steps during which nothing but movement of groups occurs (we consider these times to be unimportant). Essentially, fast-forwarding advances from one interesting time to the next, greatly improving simulation speed. As mentioned earlier, this method would also work with a separate implementation of any pathfinder, as long as that pathfinder provides a series of waypoints as orders to our groups. In a more complex domain with obstacles, more frequent re-examinations of the game state, and less abstraction, fast-forwarding is likely to be less effective since times of interest will occur closer to each other. Thus, to keep fast-forwarding effective, the abstraction model should be kept as abstract as possible.

4.2 Experimental Setup

There are several different RTS games on the market today. Each game has different units, different abilities, different resources, and other variations. Because we are creating an algorithm that should work in general, i.e., for all types of RTS games, our scenarios will only have elements that are common among all of them.

Scenario

A *scenario* is an experimental run involving a description of the initial setup of the map paired with two particular AI players controlling each side. All of our scenarios consist of bases and groups of units. Bases only have two attributes: position and hitpoints. These are abstractions of actual RTS game bases, which are typically composed of multiple buildings. Groups are composed of several units of possibly varying types. Units have the following properties: speed, attack power, armor, attack rate, position, attack range and hitpoints. Units are treated as individuals inside a group in all respects except for move speed. In this case, groups move at the speed of the slowest of its units.

Each scenario we create is symmetric geometrically, and unit-wise, giving no advantage to any player. Although this symmetry does not accurately represent real world RTS games, it does decrease result variance, which is useful for our experimentation.

Every map used has a continuous coordinate system with infinite size. There are two reasons for this choice: to avoid unnecessary collision-checking and to encourage the development of policies that are independent of the map size. However, bases and groups from the same player start fairly near each other to better approximate real world scenarios.

4.3 Initial Experiments

This section explores the effectiveness of our simulation-based planning algorithm (RTSplan) when applied to the abstract RTS game previously described.

We ran several tournaments to first determine the best evaluation function to use and then to compare the simulation-based policy to single static policies. Games were run concurrently on several computers.

To make the experimental results independent of specific hardware configurations, the simulator used an internal clock. Thus, processor speed did not affect our experimental results. However, to do this we had to slightly modify our main execution loop, because we could no longer use an execution time limit for interleaving world execution and planning. Instead, we use parameter `pairs_per_interval`, which specifies how many entries in the payoff matrix are calculated before allowing the world to move forward. As mentioned earlier, we include a flag in our simulator which allows the choice between "true" real-time mode, and the above-mentioned mode which allows for reproducible and hardware independent results for our experimental testing.

All references to *seconds* in this section are to the simulator's internal clock. Seconds in our case are not related in any way to real-world seconds. We use them merely because the speed of the groups and other attributes are specified in this time reference.

Parameters

All of our experiments have the following parameters in common, with some experiments having additional parameters that will be discussed when they are applicable:

1. `simulation_length`: This parameter sets how many simulator seconds we *fast-forward* into the future before evaluating the given state. When set to a large value, simulations are likely to end early when the game is finished due to a victory by either side. In the reported experiments this value is set to 1000 seconds, thus effectively allowing all simulations to run until the game ends.

2. `max_simulation_time`: This parameter sets the amount of real simulator seconds that are allowed to pass before we determine a winner based on the tiebreaker criterion. The value is set to 1000 seconds as well, meaning that it is likely that only true stalemates will be subject to the tiebreak procedure.

3. `pairs_per_interval`: This parameter determines how many pairs of competing policies we run before the world time advances.

4. `time_increment`: This value determines by how much time the world advances during every interval (time between each tick in Figure 4.2). This parameter is set to 0.1 seconds which means that time in our simulation advances by 0.1 seconds for every number of policy pairs we simulate. This is specified by `pairs_per_interval`).

In our initial experiments, we use either sets of 50 or 100 maps which are similar to the map in Figure 4.4. The map in Figure 4.4 is a snapshot of only a part of the total scenario in the middle of the game. All of our initial experiments use scenarios of 4 bases per player, with each base being defended by one group of 20 tanks. We use the large numbers of tanks in order to try and make sure our results generalize to regular RTS games which often have similar numbers of units.

Evaluation Functions

The evaluation function is traditionally something that is designed by experts. Our algorithm simulates all the way to the end of the game, or at least significantly far into the future in the case where both policies end up in a stalemate situation. Thus, the evaluation function of the game state can be fairly simple, because we are simulating to the end of the game in the majority of cases.

The game state evaluation function used in our experiments was chosen by testing several candidates and determining the function that was most suitable. Each candidate function ran against static opponents that used the policies described earlier. Thus, each candidate evaluation function was run vs. every policy, for every one of the 50 randomly generated symmetrical maps. Because we had 8 predefined policies, there were a total of 400 separate games played by each candidate. Each evaluation function used a basic RTSplan player, and the `pairs_per_interval` was set to 8. This meant that the RTSplan player experienced a 0.8 ($64/8*$ `time_increment`) second delay in its reaction time, because we had a total of 64 entries to compute the payoff matrix. In this experiment, we awarded 2 points for a win, 1 point for a tie, and 0 for a loss. The win rate is based on the total number of possible points obtained.

The first candidate evaluation function was just a simple function that returned either 100 for a win, -100 for a loss, and 0 for a tie. Its win rate against the static policies was 70.4%. Our second candidate evaluation function added a time parameter, which gave a slight bonus to quick wins, and to slow losses. Thus, it prefers to win quickly, and if losing, to prolong the game as long as possible. Its win rate was 75.2%. Finally, our last evaluation function added a material difference bonus. This means that preserving our units and destroying enemy units leads to a higher score. This modified evaluation function obtained a win rate of 80.7%. Due to the significant improvement, it is the one used in all of our further experiments.

4.3.1 RTSplan Experimental Results

After we had obtained an evaluation function by comparing the performance of several candidates in conjuction with RTSplan against the static policies, we ran RTSplan against different opponents, with varying parameters, in several further experiments in order to judge its performance.

RTSplan vs. Other Policies

First, we tested the performance of our basic RTSplan player without opponent modelling vs. our individual static policies, and also against a “random” player

(Random) that switched randomly between policies every 5 game seconds. This was run over a set of 100 randomly generated symmetrical maps, with the basic RTSplan player and the Random player competing against all 8 policies one at a time for every map. Thus, there were 800 games played for every player match-up. Although the Random player did not stick to one static policy, we still played it 8 times for every map, just like for the static player. The only difference was which policy the Random player played in the first 5 seconds, before it randomly switched.

Due to this fact, the results in Table 4.1 are expressed in terms of win, loss and tie percentages. They show that overall, our basic RTSplan player beats individual static policies. This is despite the fact that it operates at the same 0.8 second delay as we have seen in the evaluation function experiments. However, a closer examination of the results show that while RTSplan performs well against most opposing policies, it performs poorly against a few others. This will be explored later in this chapter.

The results for the Random player are interesting as well. It gets beaten fairly handily by both the static policies and the RTSplan player, which is not surprising, because it essentially switches policies blindly, while even the static player at least has a policy which knows how to play out the entire game. However, it still defeats the RTSplan player 15% of the time.

Because it switches policies every 5 seconds, the final policy for the Random player is a mix of our 8 defined policies, which is similar to what the RTSplan player does. Our forward simulations do not currently allow for the switching of policies in the middle of simulations, and thus they cannot foresee some of the erratic movements of the Random player. This means that the Random player can

Table 4.1: Different Player Comparison

Row vs. Col (W-L-T) %	Random	Static	RTSplan
Random	-	22.0 - 77.8 - 0.2	15.0 - 84.0 - 1.0
Static	77.8 - 22.0 - 0.2	-	21.9 - 78.1 - 0.0
RTSplan	84.0 - 15.0 - 1.0	78.1 - 21.9 - 0.0	-

get "lucky" and catch our RTSplan player off guard with an unforeseen move.

RTSplan vs. Individual Policies

Although we have shown that the basic RTSplan player can beat the individual policies overall, it is also useful to know its performance against each static policy specifically. These results are shown in Table 4.2.

From these results, it is clear that RTSplan player soundly defeats every policy with the exception of the policies that are aggressive, especially the Hunter policy. Thus, we need to determine how well Hunter performs against all the *other* policies to determine if Hunter is the dominant policy. These results can be seen in Table 4.3.

The results show that Hunter is soundly beaten by any policy that forms one large mass of units at the outset. This means that it is not the dominant policy and that some other factor is causing the RTSplan player to lose to Hunter because the RTSplan player performs much better *overall*, yet loses to Hunter.

At first, it appears likely that the delay that our RTSplan player experiences due to its staggered payoff matrix computation process could be the cause. We performed an experiment to test this, where we gave the RTSplan player no delay. However, Table 4.4 shows that although the delay does hamper the performance of the RTSplan player, it is not the only cause. Even in this ideal scenario, RTSplan still loses to Hunter. This result will be explored later in this chapter.

It may also seem like we are double-counting the results for Mass Attack and Spread Attack in Tables 4.2, 4.3 and 4.4. However, this is not the case. The policies with base and unit preferences are in reality similar, yet still very different. In our particular scenarios however, they end up yielding the same results. It is likely they would yield different results in a different scenario setup.

RTSplan vs. MinMax and MaxMin

Our RTSplan simulation player generally defeats single policies. To address the question of how important policy randomization is in our game, we created two other players. These players use simulation but treat the games as alternating move games, in which moves of the first player are made public and the second player

Table 4.2: RTSplan Player vs. Specific Policies

Policy	Wins	Losses	Ties
Null	100	0	0
Join Defense	98	2	0
Mass Attack (base)	98	2	0
Mass Attack (units)	98	2	0
Spread Attack (base)	51	49	0
Spread Attack (units)	51	49	0
Half Defense-Mass Attack	98	2	0
Hunter	31	69	0

Table 4.3: Hunter vs. Specific Policies

Policy	Wins	Losses	Ties
Null	100	0	0
Join Defence	7	93	0
Mass Attack (base)	7	93	0
Mass Attack (units)	7	93	0
Spread Attack (base)	69	31	0
Spread Attack (units)	69	31	0
Half Defense-Mass Attack	7	93	0

Table 4.4: RTSplan Player (no delay) vs. Specific Policies

Policy	Wins	Losses	Ties
Null	99	1	0
Join Defense	97	3	0
Mass Attack (base)	97	3	0
Mass Attack (units)	97	3	0
Spread Attack (base)	55	44	1
Spread Attack (units)	55	44	1
Half Defense-Mass Attack	97	3	0
Hunter	45	55	0

can respond to them. We call these players MinMax and MaxMin.

Naturally, we expected the RTSplan player to defeat both the MinMax and MaxMin players, because the game we consider is a simultaneous move game. To see this, consider Rock-Paper-Scissors. In an alternating and public move setting the second player can always win. We ran the MinMax and MaxMin players against each other on 100 randomly generated symmetric maps. `pairs_per_interval` was set to 64, allowing the full payoff matrix to be computed before advancing time in the simulation. The results can be seen in Table 4.5.

As expected, the MinMax and MaxMin players were almost equivalent, while it is clear that the RTSplan player is the better player by a small margin. The large number of ties is expected as well, because when both players reach the Nash-equilibrium point, neither player will have incentive to deviate from their current policy. In reality, this means that the game ends in a stalemate, where the groups of both players are stationary or in a state of oscillation.

4.3.2 Results of RTSplan with Additional Policies

The previous experimental results showed that the RTSplan player was better *overall* versus individual static policies, however, they also showed that it performs fairly poorly against the more aggressive policies such as Hunter and SpreadAttack.

As discussed earlier, the strength of RTSplan is highly dependent on the number and diversity of policies available to it, and thus we introduce the following two policies to improve the RTSplan player.

- **Attack Least Defended Base**. This policy first creates one large army in the

Table 4.5: Simulation Players Comparison

Row vs. Col (W-L-T)	MinMax	MaxMin	RTSplan
MinMax	-	6-8-86	0-21-79
MaxMin	8-6-86	-	6-19-75
RTSplan	21-0-79	19-6-75	-

same manner as Mass Attack, and then this army looks for the least defended base, which is the base that is furthest away from its friendly forces, and attacks that base. The least defended base is reconsidered periodically, in case the opponent shifts forces to defend the target base.

- **Harass**. This policy harasses the enemy, but never engages in direct combat unless forced to do so. First, armies of two nearby groups are formed, and each army is sent to attack the nearest enemy base. However, if an enemy army gets near any of our harassing armies, that army retreats in the direction of the nearest friendly base. Once it is sufficiently far enough (in this implementation, about twice the range of the enemy's weapons) from the enemy however, it proceeds to attack the nearest base once again.

Results presented in Tables 4.6 and 4.7 show that the addition of these two policies does not introduce any dominant policy. Instead it creates a better mix of policies, where every policy can be countered by at least one other. This could benefit the RTSplan player, since it can find a counter to every policy.

In theory, the RTSplan player should perform better overall if it has more policies to choose from. Having a set of policies that each have a way to be countered benefits the RTSplan player, because it has the opportunity to use the appropriate countering policy.

Table 4.8 shows that the RTSplan player does improve over the original, however, only slightly. The original 8 policy player wins 78.1% of the time, while the new RTSplan with 10 policies wins only 78.3% of the time, although game logs show that when it loses, it's a closer contest.

Further examination from game logs where the RTSplan player lost made it clear what was happening. Essentially, the root of the problem is that the RTSplan player gives too much credit to its opponents. Because it has no knowledge about the opponent, it must give it the benefit of the doubt and assume that they will use the policy most detrimental to the RTSplan player.

Unfortunately, this sometimes leads to the situation where RTSplan sees that if the opponent executes a particular policy at that specific time, there is nothing

Table 4.6: Harass vs. Specific Policies

Policy	Wins	Losses	Ties
Null	2	0	98
Join Defence	100	0	0
Mass Attack (base)	100	0	0
Mass Attack (units)	100	0	0
Spread Attack (base)	38	59	3
Spread Attack (units)	38	59	3
Half Defense-Mass Attack	100	0	0
Hunter	49	51	0
Attack Least Defended	100	0	0

Table 4.7: Attack Least Defended vs. Specific Policies

Policy	Wins	Losses	Ties
Null	100	0	0
Join Defence	13	8	79
Mass Attack (base)	6	94	0
Mass Attack (units)	6	94	0
Spread Attack (base)	0	100	0
Spread Attack (units)	0	100	0
Half Defense-Mass Attack	6	94	0
Hunter	93	7	0
Harass	0	100	0

Table 4.8: RTSplan vs. New Policy Set

Policy	Wins	Losses	Ties
Null	100	0	0
Join Defence	96	4	0
Mass Attack (base)	99	1	0
Mass Attack (units)	99	1	0
Spread Attack (base)	38	62	0
Spread Attack (units)	38	62	0
Half Defense-Mass Attack	99	1	0
Hunter	46	54	0
Attack Least Defended	99	1	0
Harass	70	28	2

that the RTSplan player can do to prevent a loss. Game logs show that this occurs only sporadically, however, the defeatist attitude from this calculation leads to *indecision*-like behaviour, where RTSplan is essentially frozen in place, trying to decide whether it can get out of a sure loss or not. This delay usually means that the situation only gets worse, which eventually leads to defeat.

A way to deal with this problem is to limit the policy set we use for our opponent by only considering policies that the opponent appears to be playing. Opponent modelling is a natural way to limit this policy set.

4.3.3 RTSplan-O Experimental Results

The previous section showed that although RTSplan plays well in general, it does have a weakness in that it gives too much credit to its opponents. This section shows that the addition of opponent modelling can remedy this problem, at least for when dealing with known policies.

RTSplan-O vs. Individual Policies

Once opponent modelling was added to the RTSplan player, the exact same map set was run once again with the RTSplan-O player versus the entire policy set.

Table 4.9 shows that the RTSplan-O player is superior to the original RTSplan player. The aggressive policies still perform fairly well, however, RTSplan-O now clearly beats *every* static policy. These results show that the problem discussed in the previous section can be overcome with opponent modelling. This result is unsurprising, since the opponents are static and are already present in our policy set. The few losses that still do occur are due to the delay that the RTSplan player still experiences.

As mentioned in Chapter 3, there are two parameters that affect the performance of opponent modelling: prediction cutoff ratio (f) and prediction interval (t).

Experiments on 1000 maps were performed to get an idea of what values were appropriate for the specified scenario setup. Table 4.10 shows that for $f = 0.1$, $t = 2.0$ yields the best results. Table 4.11 shows that for $t = 2.0$, the results are best with $f = 0.5$. Thus, Table 4.9 was run with $t = 2.0$ and $f = 0.5$. However, because

Table 4.9: RTSplan-O vs. Policy Set ($t = 2.0, f = 0.5$)

Policy	Wins	Losses	Ties
Null	100	0	0
Join Defence	97	1	2
Mass Attack (base)	99	1	0
Mass Attack (units)	99	1	0
Spread Attack (base)	92	8	0
Spread Attack (units)	92	8	0
Half Defense-Mass Attack	99	1	0
Hunter	95	4	1
Attack Least Defended	100	0	0
Harass	83	8	9

Table 4.10: RTSplan-O vs. Policy Set ($f = 0.1$)

t	Wins	Losses	Ties
1.0	917	79	4
2.0	891	100	9
5.0	863	130	7
10.0	870	126	4

Table 4.11: RTSplan-O vs. Policy Set ($t = 2.0$)

f	Wins	Losses	Ties
0.1	935	63	2
0.2	954	36	10
0.5	956	32	12
0.75	945	48	7
1.0	933	63	4

the f and t parameters are not independent, the values used are merely educated guesses. Finding the optimal values for both is time consuming and not explored in this thesis. One possible solution is to learn the values in an automated way, similar to what is done in many common learning algorithms.

RTSplan-O vs. RTSplan

Now we turn to the question of whether our new RTSplan-O player is better than the original RTSplan player. Theoretically, it should not be any better because the RTSplan player is theoretically unexploitable. 1000 maps were used in this experiment, and as expected, the difference between the two players was not statistically significant (one sided binomial, p=0.964293, n=1000, $\alpha = 0.05$), with the new RTSplan-O player winning 527 games, losing 471 and tieing 2.

RTSplan and RTSplan-O vs. Unknown Policies

Dealing with an opponent that is limited to *our* policy set is not the best way of evaluating the strength of RTSplan. In real game scenarios, it is unlikely that the opponent will be following a policy in our policy set.

Because it is not feasible to design an experimental setup with a human opponent, we instead use one of the policies in our policy set for our opponent, but we make RTSplan blind to its existence. Essentially, the policy our opponent uses is never added to the *active policies* vector, effectively preventing RTSplan from reasoning about it.

We examined the performance of RTSplan and RTSplan-O when blinded to each policy. The results are shown in Table 4.12 and Table 4.13. These results show that without opponent modelling, RTSplan's performance does not change significantly, even when dealing with an unknown policy. In fact, due to the "giving too much credit" problem discussed earlier, the performance even increases a little bit against some policies. This is because RTSplan can't give the benefit of the doubt to our opponent about a policy we know nothing about. This shows that RTSplan can deal with unknown policies, and perform acceptably. However, it is also clear that the performance of RTSplan-O suffers significantly when dealing

Table 4.12: RTSplan vs. Unknown Policy

Unknown Policy	Wins	Losses	Ties
Null	95	5	0
Join Defence	97	3	0
Mass Attack (base)	98	2	0
Mass Attack (units)	98	2	0
Spread Attack (base)	61	39	0
Spread Attack (units)	61	39	0
Half Defense-Mass Attack	100	0	0
Hunter	41	59	0
Attack Least Defended	100	0	0
Harass	45	53	2

Table 4.13: RTSplan-O vs. Unknown Policy ($t = 2.0$, $f = 0.5$)

Unknown Policy	Wins	Losses	Ties
Null	95	5	0
Join Defence	99	0	1
Mass Attack (base)	100	0	0
Mass Attack (units)	100	0	0
Spread Attack (base)	92	8	0
Spread Attack (units)	92	8	0
Half Defense-Mass Attack	100	0	0
Hunter	37	63	0
Attack Least Defended	100	0	0
Harass	49	47	4

with unknown policies. This result is not surprising because RTSplan-O cannot exploit a policy it does not recognize. Ideally, RTSplan-O would have a larger set of policies, and unknown policies could be fitted better to the nearest known policy and exploited accordingly. Essentially, the performance RTSplan-O depends on the unknown policy and how "similar" it is to the remaining policies in the policy set.

Currently, because the policy set is fairly small, no policy closely resembles the unknown policy, and thus no exploitation can be performed. The only exceptions are the MassAttack and SpreadAttack policies which have nearly identical similar policies. This reflects the significant improvement in performance of SpreadAttack in Table 4.13. However, when *both* of the SpreadAttack policies are removed from the *active policies* vector, the result is significantly worse, with 43 wins, 55 losses, and 2 ties, because there is no similar policy the opponent modelling can recognize and exploit.

RTSplan-O Policy Set Reduction

One result that needs to be examined is the effectivess of RTSplan-O at reducing the size of the opponent's policy set. In other words, how accurately we can model our opponent. The results of this are shown in Table 4.14. The first column represents the percentage of the opponent's original policy set that remains after we apply opponent modelling against the given policy. The second column shows the same percentage, however, in this case, the opponent's policy is treated as an unknown policy in the same way as was done for Table 4.13.

The results show that in almost all cases we get a significant reduction in the opponent's policy set ("active" policies is small). The only exception is where our opponent does nothing, and we are blind to that option being available to our opponent. This is because no other policy resembles the `null` policy, and thus, our opponent modelling process is forced to add all available policies to the "active" policy set. Conversely, if we know of the `null` policy, then our opponent modelling is perfect, and reduces the opponent's policy set by the maximum value (10% of the normal is perfect in this case, since we reduce the policy set from 10 to 1).

Unsurprisingly, the results also show that the more different a policy is from

the other policies in the policy set, the better our opponent modelling will do if we know about that policy, since it can be more easily identified. However, that same attribute of the policy also makes opponent modelling less effective if we don't have it in our policy set, since no other policy closely matches the missing policy. This suggests that the best way to deal with modelling unknown policies is to have a large and diverse policy set in the hopes that one of the known policies will resemble the policy being played by the opponent.

4.3.4 Execution Times

Simulation Execution Time

For our algorithm to be useful in a real RTS game setting, our computations must be able to conclude in a reasonable amount of time. Table 4.15 shows the executions times, with various percentiles, for the time it takes to perform one single forward simulation. Different scenarios sizes, numbers of policies, and the effects of opponent modelling are shown. All results were executed on a dual-processor Athlon MP 2000+ CPU, with memory ranging from 2GB to 3GB, although $<$ 100KB of memory is used by our program.

Even though some slight spikes in performance are exhibited, as can be seen in the max value, generally the execution time of a simulation is fairly low. These results show that even while computing several forward simulations every frame, we can still run at a real-time frequency, with the number of simulations run per frame determined by available CPU time. These numbers are dependent on the `simulation_timeout` parameter, as well as the `pairs_per_interval` parameter. Lowering these two parameters will result in faster execution times, although at the cost of some performance. Also, if the execution times are not acceptably low for an RTS game, it is always possible to simulate a shorter time into the future. Currently, we simulate the entire game, and lowering this threshold should increase execution time significantly.

Fast-forwarding Computational Savings

We mentioned earlier that fast-forwarding greatly speeds up forward simulation time. Table 4.16 shows the comparison between the average length of time it takes to play out one pair of policies using fast-forwarding, and the average length of time it takes to play out the same policies as a normal simulation with `time_increment` set to 0.1. In order to ensure that both the fast-forwarding and the normal simulations yielded the same game result, we only examined policies that do not periodically look at the game state since the frequency at which the the game state is examined would differ between the two methods.

We ran each policy against all the other policies in our test policy set, and then calculated the average length of time for both simulation methods. This is done in order to examine whether the choice of the policy has any effect on the results.

The results in Table 4.16 show that fast-forwarding is significantly faster than if the simulation were advanced normally, 0.1 game seconds at a time. In fact, fast-forwarding is on average over 800 times faster than regular simulations. This is mainly because we drastically reduce the number of game state recalculations, thus reducing the total computation time used for collision checking, object movement, etc., all of which must be performed every game state update.

The results show that fast-forwarding is essential for allowing RTSplan to run effectively and in real-time. The only downside of fast-forwarding is that it does not allow policies to re-examine the state continuously, but rather periodically, as specified by the policy itself. However, in this case the advantage of being able to run forward simulations at a very high speed clearly outweighs the disadvantages, as long as fast-forwarding can be kept effective by keeping the abstraction level fairly simple.

Table 4.14: Opponent Modelling Policy Set Reduction ($t = 2.0$, $f = 0.5$, 50 maps)

Policy	Known (% of normal)	Unknown (% of normal)
Null	10.0	100.0
Join Defence	29.11	32.95
Mass Attack (base)	55.90	55.13
Mass Attack (units)	55.90	55.13
Spread Attack (base)	42.17	34.31
Spread Attack (units)	42.17	34.31
Half Defense-Mass Attack	55.90	58.66
Hunter	21.66	81.24
Attack Least Defended	40.27	69.03
Harass	25.88	54.34

Table 4.15: Execution Times (milliseconds) Percentiles and Max Time

Map Size (# of policies)	10th	25th	50th	75th	90th	Max
3 bases (8)	1.13	2.08	3.34	5.42	9.16	71.39
5 bases (8)	2.26	4.72	7.83	21.93	38.92	194.85
4 bases (10)	18.62	38.30	67.01	102.12	132.22	225.57
4 bases (10) (modelling)	5.19	11.50	28.21	50.15	95.4	220.3

Table 4.16: Fast-forwarding time comparison (seconds)(50 maps)

Policy	Fast-forwarding Time	Normal Simulation Time
Null	0.033	25.20
Mass Attack (base)	0.040	20.65
Mass Attack (units)	0.016	20.13
Spread Attack (base)	0.022	11.55
Spread Attack (units)	0.015	11.31
Half Defense-Mass Attack	0.019	22.64

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis represents preliminary work in the area of simulation-based planning in RTS games. First, we introduced an abstraction model for RTS games that makes simulation-based planning in the domain feasible. The algorithm RTSplan, which is based on a combination of forward simulations and Linear Programming to obtain a Nash-optimal solution is presented. Furthermore, we introduced the concept of *fast-forwarding*, which significantly increased the speed of forward simulation, a necessity for the RTSplan algorithm. In fact, fast-forwarding has so far eliminated the need for a sophisticated evaluation function because we can simulate to the end of the game (although this may only be due to the simplicity of the scenarios). Finally, we introduced an opponent modelling extension to the RTSplan algorithm (RTSplan-O) that strives to exploit predictable players and greatly increases the performance of the basic RTSplan algorithm when used against the individual policies in our policy set.

Initial results are promising. Based on the current world state, RTSplan chooses between available policies to defeat its opponents, and even plays well against opponents that execute unknown policies. RTSplan is fairly computationally intensive, however, it can be made to work seamlessly in real-time, albeit at some cost to performance. A strength of RTSplan is that its performance is closely related to the variety of actions available to it. The more diverse the policies, and the more action space that is covered, the better the performance will be.

5.2 Future Work

There are several potential areas of improvements and extensions for RTSplan and RTSplan-O. First, RTSplan needs to be tested in a true RTS game environment such as ORTS [23]. This requires the development of an interface that allows for communication between the game's pathfinder and RTSplan, as well as an abstraction scheme that can convert real RTS game world data into the abstract form used by RTSplan. Furthermore, a more sophisticated, and game dependent combat model should be developed, as the one currently used by RTSplan is quite simplistic.

More policies should also be developed, as the current set is still fairly limited in scope and is not representative of all the common policies used in RTS games today. This could lead to a more computationally intensive process, so a distributed computing approach should be explored, especially because RTSplan lends itself nicely to parallelization.

Developing more complex scenarios that use different bases and unit types should be developed and tested, along with creating new policies that take advantage of these new unit types. This would be a step towards better representing actual RTS game abstractions.

Further testing needs to be done on the performance of RTSplan. Specifically, RTSplan should be tested against a finely-tuned AI script. Unfortunately, none was available at the time of writing. RTSplan should also be tested against human opponents. Some preliminary work has been done on this using a built-in human interface, however, no rigorous experimental setup has been set up.

Some speed optimizations could also be made to the underlying simulator. For example, currently, the simulator uses a naive approach to collision detection between objects which does not scale well with size. The execution times should be lowered in general to work better in true RTS games where a lot of the CPU time is used for other tasks such as pathfinding, combat simulation and graphics.

As discussed earlier, finding optimal values for the various parameters used in this thesis could lead to better results. The current parameters are merely educated guesses based on some simple experimentation.

Allowing forward simulations to change the policy in the middle of simulation based on either a time limit or other condition should also be explored. We have an implementation of this, however, execution time is increased exponentially because we are doing an additional full payoff matrix computation for every entry in the original payoff matrix. This results in drastically more forward simulations that need to be performed. Because execution times are currently much too slow to be considered for a real-time setting, researching methods into increasing the execution speed and/or pruning the number of simulations to be done could allow this to become feasible in the future.

Finally, further exploration into the adaptation of RTSplan to work in an imperfect information setting, apart from simultaneous moves, needs to be performed. Currently, RTSplan works only with perfect information, and assumes that a separate "scout AI" deals with incomplete information and provides information to RTSplan in the form of known and predicted enemy locations. The validity of this assumption should be tested in an RTS game setting, requiring the development of a separate scout AI, and if it is found to be unacceptable, modification of our algorithms to allow them to deal with an imperfect information environment may become necessary.

Bibliography

[1] T. Bersano-Begey. Multi-Agent Teamwork, Adaptive Learning, and Adversarial Planning in Robocup Using a PRS Architecture. `citeseer.ist.psu.edu/53896.html`.

[2] 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.

[3] Blizzard. Starcraft. `http://www.blizzard.com/starcraft`, 1998.

[4] A. Botea, M. Mueller, and J. Schaeffer. Near Optimal Hierarchical Pathfinding. In *Journal of Game Development 1*, pages (1):7–28, 2004.

[5] B. Bouzy and B. Hemlstetter. Monte Carlo Go Developments. In *Advances in Computer Games 10*, pages 159–174. Kluwer Academic Press, 2003.

[6] M. Buro. Solving the Oshi-Zumo Game. In *Proceedings of the Advances in Computer Games Conference 10*, pages 361–366. Graz, 2003.

[7] M. Chung, M. Buro, and J. Schaeffer. Monte Carlo Planning in RTS Games. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence in Games*, pages 117–124, New York, 2005. IEEE Press.

[8] K. Currie and A. Tate. O-Plan: the Open Planning Architecture. *Artificial Intelligence*, 52:49–86, 1991.

[9] D. C. Pottinger. Terrain Analysis in Realtime Strategy Games. In *Game Developers Conference Proceedings*, 2000.

[10] A. Davidson, D. Billings, J. Schaeffer, and D. Szafron. Improved Opponent Modeling in Poker. In *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI'2000)*, pages 1467–1473, 2000.

[11] E. Dybsand. Goal Directed Behavior Using Composite Tasks. In *AI Game Programming Wisdom 2*, pages 237–245. Charles River Media, 2003.

[12] Ensemble Studios. Age of Empires. `http://www.microsoft.com/games/empires/default.htm`, 1997.

[13] K. Erol, J. A. Hendler, and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. In *Artificial Intelligence Planning Systems*, pages 249–254, 1994.

[14] R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence*, pages 2:189–208, 1971.

[15] S. Gelly and Y. Wang. Exploration Exploitation in Go: UCT for Monte-Carlo Go. In *NIPS-2006, Online trading between exploration and exploitation, Whistler Canada*, 2006.

[16] M. Ginsberg. GIB: Steps Toward an Expert-level Bridge-playing Program. In *International Joint Conference on Artifical Intelligence*, pages 584–589, 1999.

[17] R. M. Jensen. *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains*. PhD thesis, 2003.

[18] T. Kent. Multi-Tiered AI Layers and Terrain Analysis for RTS Games. In *AI Game Programming Wisdom 2*, pages 448–455. Charles River Media, 2003.

[19] J. Lee. *A Simulation-Based Approach for Decision Making and Route Planning*. PhD thesis, 1996.

[20] J. Lee and P. A. Fishwick. Real-Time Simulation-Based Planning for Computer Generated Force Simulation. Technical Report TR94-034, 1994.

[21] J. Lee and P. A. Fishwick. Simulation-based Real-time Decision Making for Route Planning. In *WSC '95: Proceedings of the 27th conference on Winter Simulation*, pages 1087–1095, 1995.

[22] J. Lee and P. A. Fishwick. Simulation-based Planning for Multi-Agent Environments. In *In Proceedings of Winter Simulation Conference'1997*, pages 405–412, 1997.

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

[24] J. Nash. Equilibrium Points in N-person Games. In *Proceedings of the National Academy of the USA 36(1)*, pages 48–49, 1950.

[25] D. S. Nau, T. C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, december 2003.

[26] J. Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Game Developers Conference 2006*, 2006.

[27] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[28] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[29] E. Sidran. The Current State of Human-Level Artificial Intelligence in Computer Simulations and Wargames. `http://www.gilgameshcontrite.com/Computer_AI/`, 2003.

[30] F. Southey, W. Loh, and D. Wilkinson. Inferring Complex Agent Motions from Partial Trajectory Observations. In *Proceedings of IJCAI*, pages 2631–2637, 2007.

[31] N. Sturtevant, M. Zinkevich, and M. Bowling. ProbMaxn: Opponent Modeling in N-player Games. In *National Conference on Artificial Intelligence (AAAI)*, pages 1057–1063, 2006.

[32] G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM Archive*, 38(3):58–68, 1995.

[33] P. Tozour. Influence Mapping. In *Game Programming Gems 2*, pages 287–297. Charles River Media, 2001.

[34] P. Tozour. Strategic Assessment Techniques. In *Game Programming Gems 2*, pages 298–306. Charles River Media, 2001.

[35] P. Tozour. Using a Spatial Database for Runtime Spatial Analysis. In *AI Game Programming Wisdom 2*, pages 381–390. Charles River Media, 2004.

[36] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen 100*, pages 295–320, 1928.

[37] N. Wallace. Hierarchical Planning in Dynamic Worlds. In *AI Game Programming Wisdom 2*, pages 382–390. Charles River Media, 2003.

[38] Westwood. Red Alert. `http://www.ea.com/official/cc/firstdecade/us/redalert.jsp`, 1996.

[39] S. Willmott, J. Richardson, A. Bundy, and J. Levine. Applying Adversarial Planning Techniques to Go. *Theoretical Computer Science*, 252(1–2):45–82, 2001.

[40] K. Woolsey. Computers and Rollouts. `http://www.gammonline.com/members/Jan00/articles/roll.htm`, 2000.