Adversarial Planning Through Strategy Simulation

Frantisek Sailer, Michael Buro, and Marc Lanctot
Dept. of Computing Science
University of Alberta, Edmonton
sailer|mburo|lanctot@cs.ualberta.ca

***Abstract*— Adversarial planning in highly complex decision domains, such as modern video games, has not yet received much attention from AI researchers. In this paper, we present a planning framework that uses strategy simulation in conjunction with Nash-equilibrium strategy approximation. We apply this framework to an army deployment problem in a real-time strategy game setting and present experimental results that indicate a performance gain over the scripted strategies that the system is built on. This technique provides an automated way of increasing the decision quality of scripted AI systems and is therefore ideally suited for video games and combat simulators.**

Keywords: real-time planning, simulation, game theory

I. Introduction

Planning is the process of determining action sequences that when executed accomplish a given goal. Main-stream planning research focuses mostly on single-agent planning tasks without adversaries who actively try to prevent the agent from attaining its goal as well as try to achieve their own – often conflicting – goals. The presence of adversaries in addition to real-time and hidden information constraints greatly complicates the planning process. The biggest success in the area of adversarial planning has been mini-max game-tree search whose application to chess and checkers has produced AI systems on par with human experts or better. Due to the tactical nature of these board games and their relatively small branching factor, alpha-beta search can look far ahead and often secure a victory by seeing a beneficial capture earlier than human players. Many game-tree search algorithms are based on exhaustive enumeration and evaluation of future states. This precludes them from being directly applied to more complex adversarial decision problems with vast state and action spaces, which, for instance, players of modern video games are often faced with when battling opponents with hundreds of units in real-time. One idea to solve such problems is to find suitable abstractions of states and actions that allow us to approach the adversarial planning task by mini-max search in abstract space.

In this paper we investigate one of such abstractions that considers whole strategies as the subject of optimization rather than individual low-level actions. Our application area of choice is real-time strategy (RTS) games, which will be described in some detail in the next section. We then present an algorithm for strategy selection based on simulation and Nash-equilibrium [1] approximation, followed by a discussion of implementation details when applied to an RTS game army deployment task, and experimental results. A section on future work on adversarial planning in RTS games concludes the paper.

II. AI for RTS Games

One popular genre of computer games on the market today is real-time strategy (RTS) games. In a typical 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. Some examples of popular RTS games are Red Alert [2], Age of Empires [3] and StarCraft [4]. RTS games differ from classic games such as Chess, Checkers and Go, in several respects. They usually feature dozens of unit types, several types of resources and buildings, and potentially hundreds of units in play at the same time. Unlike most classic games, all players also make their moves simultaneously. Furthermore, 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 player vision which is usually limited to areas around allied units and buildings.

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 and where to find them. Players must also decide when and where to build which structures and when to train which units.
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.
3) **Army Deployment.** Once a player has built groups of units, these groups need to be given orders on what to do, e.g. defend a base, attack enemy encampment, and/or move to location.

AI systems in today's commercial RTS games are scripted. For example, there is often a precise set of instructions that the AI follows at the start of the game in order to develop its base. Once this script achieves its goal condition, the AI system will switch over to a new sequence of instructions, and start to follow them, etc. While this approach does give the AI the ability to play the game in a seemingly intelligent manner, it does have several limitations. First, the AI has a limited set of scripts, and thus its behaviour can quickly become predictable. Also, because every script needs to be created by experts and can take a long time to implement and test, developing a scripted AI system for an RTS game can be a major undertaking. Furthermore, scripts are usually

inflexible and any situation not foreseen by the script creators will likely lead to inferior game play.

To compensate for these shortcomings, current commercial RTS game AI systems are often given extra advantages in form of more resources or full knowledge of the game state. While this approach seems to be acceptable in campaign modes that teach human players the basic game mechanics, it does not represent a solution to the RTS game AI problem of creating systems that play at human level in a fair setting.

There are several reasons for which there exist no good solutions to RTS AI thus far:

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 hitpoints, move speed, attack power, and range. In contrast, Chess units each only have one attribute: their move ability. Due to the complexity of RTS game units, traditional AI search techniques such as alpha-beta search have trouble dealing with such large state spaces.
2) **Real-Time Constraint.** Tactical decisions in RTS games must be made quickly. Any delay could render a decision meaningless because the world could 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 game board in any classical game. Checkers has 32 possible positions for pieces, Chess has 64, Go has up to 361. By contrast, even if the RTS game does not happen to be in continuous space, it often has ten-thousands of possible positions a unit could occupy. Furthermore, the number of units in an RTS game is often in the hundreds.
4) **Simultaneous Moves.** Units in RTS games can act simultaneously. This presents a problem for traditional search techniques, because the actions 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 n-player games, there are currently no solutions able to run well in real-time.
6) **Incomplete Information.** RTS games are played with mostly incomplete information. Enemy base and unit locations are initially unknown, and decisions must be made without this knowledge until scouting units are sent out. Currently, there are no AI systems that can deal with the general incomplete information problem in the RTS domain. However, recent work on inferring agent motion patterns from partial trajectory observations has been presented [5]. This, in addition to results obtained for the classic imperfect information domains of bridge [6] and poker [7] and the work presented here, may soon lead to stronger RTS game systems.

Due to these properties creating a strong AI system for playing RTS games is difficult. A promising approach is to implement a set of experts on well-defined sub-problems such as efficient resource gathering, scouting, and effective targeting, and then to combine them. For example, there could be an expert 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 application we consider in this paper is army deployment. The AI system for this task does not have to worry about resource gathering, building, scouting, or even small-scale combat. Instead, it makes decisions on a grander scale, like how to split up forces and where to send them.

This paper builds on ideas presented in [8], where Monte Carlo simulation was used to estimate the merit of simple parameterized plans in a capture-the-flag game. Here, we approach this problem slightly differently, by combining high-level strategy simulation with ideas from game theory, which is described next.

III. Adversarial Planning Based on Simulation

A. The Basic Algorithm

As discussed in the introduction, abstractions are required before state-space search algorithms can be applied to complex decision problems such the ones faced in RTS games. The use of spatial abstractions, for instance, can speed up pathfinding considerably while still producing high-quality solutions. 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 strategies from which the AI can choose. Strategies are considered decision modules, functions of state to action. Consider, for instance, various ways of playing RTS games. One typical strategy is "rushing", whereby a player produces a small fighting force as quickly as possible to surprise the opponent. Another example of a typical strategy is "turtling", in which players create a large force at their home base and wait for others to attack and get defeated. It may be relatively easy to implement such strategies which, for the purpose of high-level planning, can be considered black-boxes, ie. components whose specific implementations are irrelevant to the planning process. The task of the high-level planner then is to choose a strategy 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, just like chess programs do today. Having access to a number of strategies, the question now becomes how to pick one in a given situation. Assuming we have access to the set of strategies the opponent can choose from, we can learn about the merit of our strategies by simulating strategy pairs,

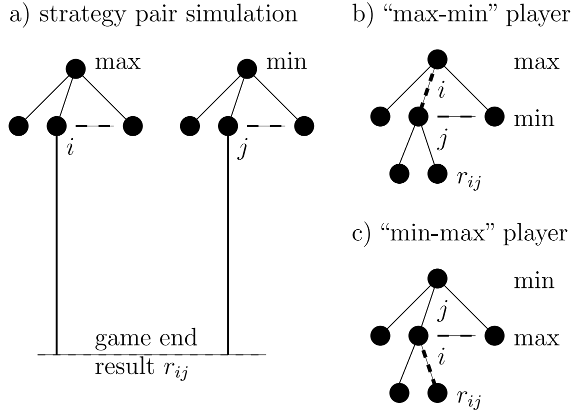


Fig. 1. a) Simulating pairs of strategies b) max-min player chooses move i which leads to the maximum value c) min-max player chooses the best counter move i

Player A (rows) vs. Player B (columns):

	R	P	S
R	0	−1	+1
P	+1	0	−1
S	−1	+1	0

Player A (rows) vs. Player B (columns):

	S_1	S_2	S_3	S_4	S_5	...
S_1						
S_2						
S_3						
S_4						
⋮						

Fig. 2. 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 simulations. S_i represents strategy i.

i.e. pitting our strategy i against their strategy j for all pairs (i, j) and storing the result r_{ij} (Figure 1a). In the simplest version, strategies would be simulated to completion or timed-out, in which case a heuristic evaluation function is necessary to estimate who is ahead.

In a zero-sum two-player setting with simultaneous moves the natural move-selection choice then would be to determine a Nash equilibrium strategy by mapping the payoff matrix r into a linear programming problem whose solution is a probability distribution over strategies. In the Nash equilibrium case, neither player has an incentive to deviate. Nash-optimal strategies can be mixed, i.e. for optimal results strategies 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 of RPS is shown in Figure 2. The Nash-optimal strategy is to choose each action uniformly at random; in particular $P(\text{Choose Rock}) = P(\text{Choose Paper}) = P(\text{Choose Scissors}) = \frac{1}{3}$. Here, the actions are instead strategies and the payoff values are obtained via results of simulations into the future. Alternatively, one could choose the mini-max rule, whereby 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 1 b) and c). In these examples player max plays move i, which leads to the best mini-max value. Only in case where there are pure Nash equilibrium strategies do the payoffs coincide. Otherwise, informing the opponent about the move choice is detrimental like in Rock-Paper-Scissors, and the Nash-optimal strategy may have advantages over max-min, or min-max, or both. The following pseudo-code summarizes the simulation approach to selecting strategies we just described:

1) Consider a set of strategies s of size n and compute each entry of the $n \times n$ result matrix by assigning strategy s_i to the simulation-based AI, and strategy s_j to its opponent, and executing these strategies 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 result payoff matrix entry r_{ij}.
2) Calculate a Nash-optimal strategy with respect to our player using the standard Linear Programming (LP) based method [9], or alternatively a min-max or max-min move.
3) In case of the Nash-optimal player, assign a strategy randomly to our player, using the probability distribution returned by the LP solver, or play the min-max or max-min move directly.
4) Repeat from step 1 as often as is desired while executing the chosen strategies.

B. Implementation Considerations

Evaluation Functions. Although the evaluation function is often something that must be designed by experts, our algorithm actually simulates all the way to the end of the game, or at least very far into the future in case both strategies end up in a stalemate situation. Because we are simulating to the end of the game in the vast majority of cases, the evaluation function can be very simple. We just check if we have won or lost, and return the result. We also consider a few other factors, and these will be discused in the experiments section.

Fast Forwarding. Our algorithm relies heavily on simulations which can be very expensive, especially if we were to simulate every single time step into the future. In order 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. The derivation of the *time of interest* is implementation specific, and will be discussed in context of our application in the next section.

Simulation Process. The main loop of our simulator looks as follows:

```
currTime = 0;
while (!isGameOver()) {
  for (int i=0; i < players.size(); ++i) {
    Strategy bestStrat;
    bestStrat = calcBestStrategy(players[i]);
    players[i].updateOrders(bestStrat);
  }
  currTime += timeIncrement;
  updateWorld(currTime);
```

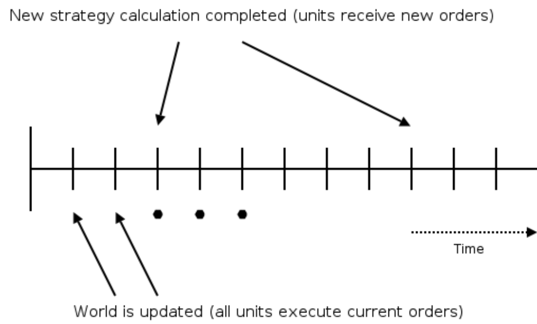


Fig. 3. The simulation timeline

```
}
determineWinner();
```

In case of simulation-based AI players we perform forward simulations to fill out the result matrix and return the new best strategy. For other player types we call the corresponding code to choose a strategy. Regardless of whether orders were changed, the world advances forward in time by the specified time increment. However, because calculating the best strategy 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 at a constant rate, even while strategy calculations are going on (see Figure 3). For example, in a typical RTS game which runs at 8 simulation frames a second, the simulator only has 1/8th of a second to perform simulation computations before the world would advance. This is enough time to compute a few entries of the payoff matrix, but 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 strategy. It is at this point that actions are being updated.

Calculation of Best Strategy. The best strategy for our Nash player is calculated in a fairly straightforward manner. First, we need to fill out the payoff matrix. Each entry in the matrix represents the result of one simulation in time between competing strategies in which a winner is found or the time limit has been reached. The basic algorithm is the following:

```
for (int i=0; i < numOurStrategies; ++i) {
  for (int j=0; j < numTheirStrategies; ++j) {
    if (!nextSimulationAllowed()){
      return notDone
    }
    // simulate the competing strategies
    r[i][j] = simulate(ourStrat[i], theirStrat[j])
  }
}
return pickStrategy(r);
```

Notice that 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 fill out the payoff matrix in order to determine the next strategy. However, in order for our player to be able to play in a real-time setting, time constraints are necessary, because filling out the entire matrix can take too long.

IV. Implementation and Experimental Setup

A. Trial Description

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, ie. for all types of RTS games, our scenarios will only have elements that are common among all of them. A *scenario* is a trial 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 bases, which are typically composed of multiple buildings. Groups are composed of several units of different 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. Furthermore, each scenario we create is symmetric (geometrically, with respect to the map), giving no advantage to any player. Although this symmetry does not accurately represent real world RTS games, it does decrease variance, which is useful for our experimentation.

Every map used has a continuous coordinate system with infinite size. There are two major reasons for this choice: to avoid unnecessary collision-checking and to encourage the development of strategies that are independent of the map size. However, opposing bases and units start near each other in order to better approximate real world scenarios. Orders given to groups are very simple. Group orders are composed of a target location, and the speed at which to travel to that location. The group then attempts to move in a straight line towards its goal from its current position restricted only by its maximum speed.

Because we are creating an AI for the general, who deals with *army* deployment, we abstract individual units into groups. 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. A human often sends out groups of units, and usually deals with individual units only when in direct combat. Our method does not deal with combat tactics, instead it has a fairly simple combat model that generally favours numerical advantage. Ideally, the AI for combat would be supplied by a separate algorithm.

It should be noted that none of our scenarios contain obstacles. Consequently, pathfinding is irrelevant in this particular application and therefore no sophisticated path-finding algorithm is included in the simulator. However, the subject of pathfinding is not ignored entirely. In fact, our algorithm is meant to work in parallel with *any* type of pathfinder. In the setup described here, a pathfinder would examine the terrain and find a path composed as a set of waypoints. These waypoints would then be passed to the AI player as orders to be executed sequentially by the groups. Essentially, pathfinding is abstracted in order to minimize

the interference with other factors inherent in strategic game play.

Victory in a scenario is achieved by one side in three different ways. Either all of the enemy's bases *or* units are destroyed before the simulation-based AI agent's, or the simulation runs past a predetermined time limit. In the latter case, the time at which to stop the 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 units. If either all the bases or units were killed at the same time, or the material is identical when time runs out, the result of the scenario is declared a tie.

B. Strategies

All of our simulations currently involve the following 8 strategies:

1) **Null**. This is more like a lack of strategy. All groups stop what they are doing, and do not move. They do however still attack any groups or bases within range.
2) **Join up and Defend Nearest Base**. This strategy gathers all the groups into one big group, and then moves this large group to defend the base that the enemy is closest to.
3) **Mass Attack**. In this strategy, all groups form one large group which then goes to attack the nearest enemy base until no enemy bases remain. There are two versions of this strategy. Given the choice of attacking a base and group, one chooses to attack the base first and the other chooses to attack the group first.
4) **Spread Attack**. In this strategy, all groups attack the nearest enemy base, and this repeats until all enemy bases are destroyed. There are two versions of this strategy; the versions are analogous to those of the Mass Attack Strategy.
5) **Half Base Defense Mass Attack**. This is a split strategy. Units are divided into two halves. One half defends their nearest bases, while the other executes the Mass Attack strategy.
6) **Hunter**. In this strategy, groups join with their nearest allied groups in order to make slightly larger combined groups. After the joining, all of these newly formed groups join into one large group which attacks the nearest enemy group.

Note that strategies which require examination of the game state (for example, to determine nearest enemy group) do so periodically. In our case, the examination is done every 5 game seconds. This choice is due to the fact that we are fast-forwarding via simulation and thus cannot examine the game state continuously.

C. Fast-Forwarding of Strategies

The simulation algorithm requires a large amount of forward simulations. More specifically, each simulation forwards all the way till the end of the game or to some point in the far future (*eg*. 1000 world seconds), Therefore, it is crucial that the simulations of future states are computed quickly. In order to meet this requirement we introduce the concept of *fast-forwarding*. The basic algorithm for fast forwarding in our RTS simulation environment is demonstrated by the following pseudo-code:

```
// 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 strategy gets to reevaluate game state
double timeoutTime = getNextStrategyTimeoutTime();
if(timeoutTime < minTime) minTime=timeoutTime;

return minTime;
```

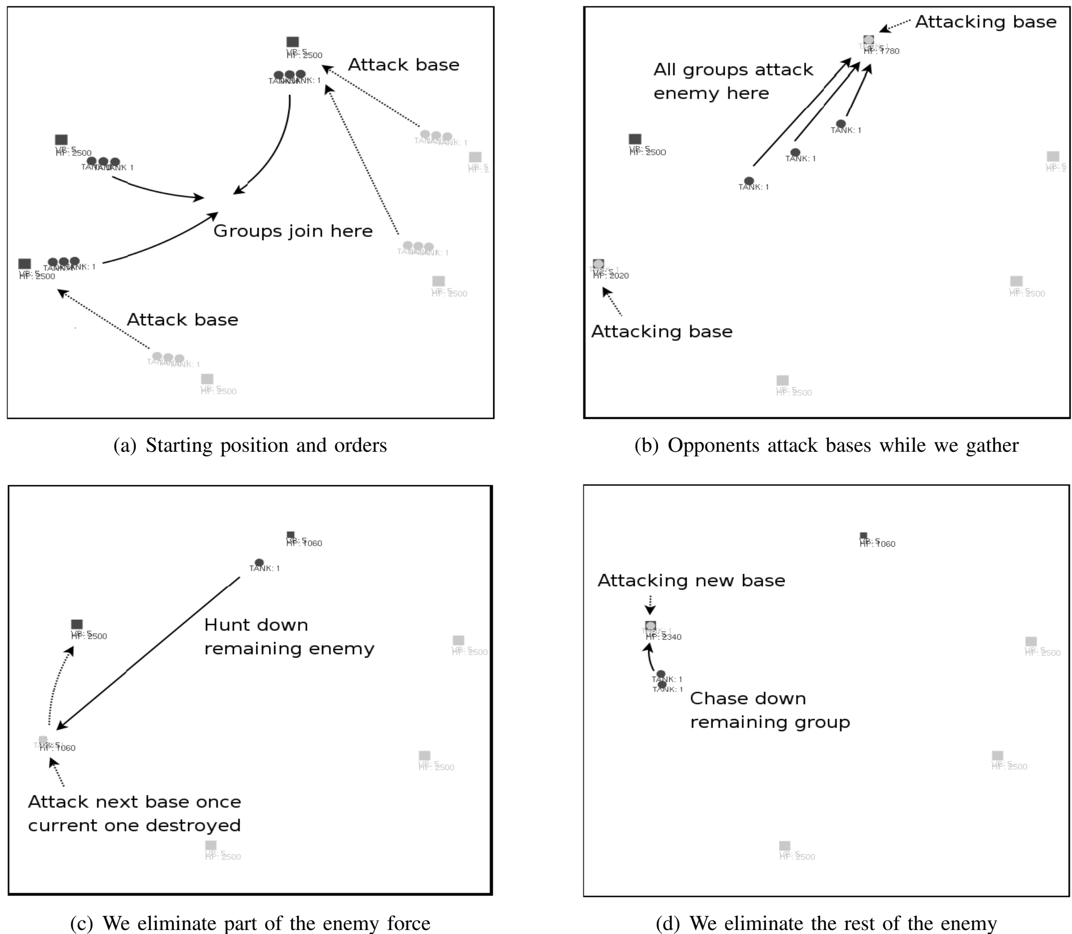
Each function is implemented differently. `nextCollideTime()` is calculated by solving a quadratic equation with input being the direction vectors of the two groups in question.

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[10], another continuous-space RTS environment. `getNextOrderDoneTime()` is a simple calculation. Because all units travel in straight lines, 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. Next, `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. Finally, the `getNextStrategyTimeoutTime()` function returns the next time that any one of the strategies in question is allowed to re-evaluate the game state in order to give out new orders if necessary.

Fast-forwarding allows the algorithm to safely skip all the times during which nothing of importance occurs. Instead, fast-forwarding jumps from one time to the next, greatly improving simulation speed. As mentioned earlier, this method would also work with a parallel implementation of a pathfinder, as long as that pathfinder provides a series of waypoints as orders to our groups.

V. EXPERIMENTS

This section explores the effectiveness of our simulation-based planning algorithms when applied to the RTS game previously described. We ran several tournaments to first determine the best evaluation function to use and then to compare the simulation-based strategy to single static strategies. Games were run concurrently on several computers.



(a) Starting position and orders

(b) Opponents attack bases while we gather

(c) We eliminate part of the enemy force

(d) We eliminate the rest of the enemy

Fig. 4. Snapshots of a typical map and the progression of a game. Light gray is a static player playing the Spread Attack strategy, dark gray is the Nash player.

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.

All references to *seconds* in this section refer to this 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.

All of our experiments have the following parameters:

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). 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 tiebreak.
3) **pairs_per_interval:** This parameter determines how many pairs of competing strategies 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 3). This parameter is set to 0.1 seconds which means that time in our simulation advances by 0.1 seconds for every number of pairs we simulate(number specified by pairs_per_interval).

In most of our experiments, we use either sets of 50 or 100 maps which are similar to the one shown in Figure 4.

The map shown is a snapshot of only a part of the total scenario in midgame. In our experiments, we use two sets of maps. The larger maps have 5 bases, with each base starting off surrounded by 4 groups of the same side. The smaller maps have only 3 bases, with 3 groups at each base.

A. Evaluation Functions

The evaluation function used in the experiments was chosen by testing several candidates and choosing the most suitable. Each evaluation function ran against a static opponent that used one of the strategies described earlier. Each map ran our evaluation function vs. every strategy, for every one of the 50 randomly generated symmetrical maps. Thus, because we had 8 defined strategies, there were a total of 400 separate games played for each evaluation function. Each evaluation function used a basic Nash player, and the pairs_per_interval was set to 8. This meant that the Nash player experienced

TABLE I

DIFFERENT PLAYER COMPARISON

Row vs. Col (W-L-T) %	Random	Static	Nash
Random	-	22-77-1	15-84-1
Static	77-22-1	-	8-73-19
Nash	84-15-1	73-8-19	-

a 0.8 ($64/8 * time_increment$) second delay in its reaction time, because we had a total of 64 entries to fill out the payoff matrix. We awarded 2 points for a win, and 1 point for a tie, and the win rate is based on the total number of possible points obtained.

The first evaluation function was just a simple evaluation function that returned a either a 100 for a win, -100 for a loss, and 0 for a tie. Its win rate was 70.4%. Our second evaluation function added a time parameter, which gave a slight bonus to quick wins, and to long losses. Thus it would prefer 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 appended a further material difference bonus. Thus, preserving our units and destroying enemy units leads to a higher score. This modified evaluation function obtained win rate of 80.7% and, due to the significant improvement, is the one used in all of our further experiments."

B. Nash vs. Strategies

Next, we tested the performance of our Nash player vs. all of our individual strategies played one at a time, and also against a "random" player (Random) that switched randomly between strategies every 5 game seconds. Once again, this was run over a set of 50 randomly generated symmetrical maps, with the Nash player and the Random player competing against all 8 strategies one at a time for every map. Thus, there were 400 games played for every player matchup. It should be noted that although the Random player did not stick to one static strategy, we still played it 8 times for every map, just like for the static player. The only difference was which strategy the Random player played in the first 5 seconds, before it randomly switched.

The results in Table I clearly show that our Nash player beats individual strategies in the majority of cases. This is despite the fact that it operates at the same 0.8 second delay as we have seen in the evaluation function experiments. In the cases when the Nash player does lose, it is most often due to the particular map situation, where a slight delay can mean the difference between victory or defeat. Because maps are symmetrical, this happens frequently.

The results for the Random player are interesting as well. It gets beaten fairly handily by both the static strategies and the Nash player, which is not surprising, because it essentially switches strategies blindly, while even the static player at least has a strategy which knows how to play out the entire game. However, it does defeat the Nash player more often than the static player does. Because it switches strategies every 5 seconds, the final strategy for the Random player is a mix of our 8 defined strategies (similar to what the Nash player does). Our forward simulations do not currently allow for the switching of strategies mid-simulation, and thus they cannot foresee some of the erratic movements of the Random player. This means that the Random player can get "lucky" and catch our Nash player off guard with an unforeseen move.

TABLE II

NASH PLAYER VS INDIVIDUAL STRATEGIES

Strategy	Wins	Losses	Ties
Null	26	0	24
Join Defense	1	2	47
Mass Attack(base)	36	12	2
Mass Attack(units)	36	12	2
Spread Attack(base)	49	1	0
Spread Attack(units)	49	1	0
Half Defense-Mass Attack	48	1	1
Hunter	46	4	0

TABLE III

JOIN DEFENSE VS INDIVIDUAL STRATEGIES

Strategy	Wins	Losses	Ties
Null	2	1	47
Mass Attack(base)	21	0	29
Mass Attack(units)	21	0	29
Spread Attack(base)	49	1	0
Spread Attack(units)	49	1	0
Half Defense-Mass Attack	48	2	0
Hunter	47	3	0

C. Nash vs. Individual Strategies

Although we know that the Nash player can beat the individual strategies overall, it is also useful to know how it performs against the static strategies individually. These results are shown in Table II.

From these results, it is clear that Nash player soundly defeats every strategy with the exception of the Join Defense strategy. Thus, we need to determine how well Join Defense performs against all the *other* strategies. These results can be seen in Table III.

In order to compare the performance of this strategy to our Nash player, we calculate the *win rate* of both. This is done by converting the results into points, with a score of 2 points of for every win, and 1 point for every loss. According to this evaluation metric, the Nash player scored a maximum of 609 out of 700 points, while the Join Defense strategy scored 579 points. Thus, the win rate of the Nash player is 87.0%, and 82.7% for the Join Defence strategy.

These results indicates that the Join Defense strategy is very strong overall. This is due to the fact that there is no proper counter-strategy to it in our strategy set. Ideally, we would want a strong counter-strategy that avoids the defended base, and attacks undefended bases. In the end, however, the Nash player still has a higher win rate than the Join Defense strategy. We suspect that the difference between these win rates would be even higher with inclusion of a proper counter-strategy.

D. Nash vs. MinMax and MaxMin

Our Nash simulation player generally defeats single strategies. In order to address the question how important strategy

TABLE IV
SIMULATION PLAYERS COMPARISON

Row v.s. Col (W-L-T)	MinMax	MaxMin	Nash
MinMax	-	6-8-86	0-21-79
MaxMin	8-6-86	-	6-19-75
Nash	21-0-79	19-6-75	-

TABLE V
EXECUTION TIMES (MILLISECONDS) PERCENTILES AND MAX TIME

Map Size	10th	25th	50th	75th	90th	Max
3 bases(each)	1.13	2.08	3.34	5.42	9.16	71.39
5 bases(each)	2.26	4.72	7.83	21.93	38.92	194.85

randomization is in our game, we created two other players that also use simulation but treat the games as alternating move games, in which moves of the first player are made public and the second player can respond to them. We call these players MinMax and MaxMin.

Naturally, we expected the Nash 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 players against each other on a set of 100 randomly generated symmetric maps of 3 bases per player, with 3 groups per base. **pairs_per_interval** was set to 64, thus allowing the full payoff matrix to be computed before advancing time in the simulation. The results can be seen in Table IV.

As expected, the MinMax and MaxMin players were almost equivalent, while it is clear that the Nash player is the better player.

E. Execution Times

In order for our algorithm to be useful in an RTS setting, our computations must be able to conclude in a reasonable amount of time. After all, RTS games make many other demands on the CPU. Table V shows the executions times, with various percentiles, for the time it takes to perform one single forward simulation. All results were executed on a dual-processor Athlon 1666 Mhz computer.

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 quite 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.

VI. CONCLUSIONS AND FUTURE WORK

This paper presents preliminary work on the development of simulation-based adversarial planners for complex decision domains. We have described an algorithm that uses results obtained from strategy-pair simulations to choose which strategy to follow next. Initial results show that with proper abstraction and fast-forwarding, simulation into the far future is efficient and feasible. Furthermore, in our RTS game application we have demonstrated that by approximating a Nash-optimal strategy we can produce a player that uses a mix of the strategies to consistently defeat individual strategies. Using this technique can help video game companies to improve their AI systems with minimal changes because most of these systems are already based on scripted strategies.

To determine the true potential of our approach, we need to test the performance of our Nash player against some highly complex scripted strategies, or against human players. Furthermore, we also need to perform more experiments, especially with larger sets of strategies, which will invariably result in a better player. We also intend to add opponent modelling to our framework in order to exploit our opponents (something our Nash player currently does not do). Incorporating opponent modelling in this algorithm would consist of keeping track of our opponent's moves and then matching up their actions to the set of strategies. This match-up could help determine which strategy is most likely being played and could influence adjustments to our result matrix accordingly.

We plan on tackling the problem of incomplete information with a combination of Monte Carlo sampling and the maintenance of a belief state of the locations of our enemies. Finally, in order to increase the skill of our players, we plan on looking at the feasibility of introducing *choice points* to our simulation framework. That is, points in time which allow the players to change strategies in mid-simulation instead of simply playing each strategy for each player for the entire forward simulation. This would result in a better player, but at the cost of an increased computational load.

ACKNOWLEDGMENTS

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

REFERENCES

[1] J. Nash, "Equilibrium points in n-person games," in *Proceedings of the National Academy of the USA 36(1)*, 1950, pp. 48–49.

[2] Westwood, "Red Alert," 1996. [Online]. Available: http://www.ea.com/official/cc/firstdecade/us/redalert.jsp

[3] Ensemble Studios, "Age of Empires," 1997. [Online]. Available: http://www.microsoft.com/games/empires/default.htm

[4] Blizzard, "Starcraft," 1998. [Online]. Available: http://www.blizzard.com/starcraft

[5] F. Southey, W. Loh, and D. Wilkinson, "Inferring complex agent motions from partial trajectory observations," in *Proceedings of IJCAI, to appear*, 2007.

[6] M. Ginsberg, "GIB: Steps toward an expert-level bridge-playing," in *International Joint Conference on Artifical Intelligence*, 1999, pp. 584–589.

[7] D. Billings, L. Pena, J. Schaeffer, and D. Szafron, "Using Probabilistic Knowledge and Simulation to Play Poker," in *AAAI National Conference*, 1999, pp. 697–703.

[8] 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*. New York: IEEE Press, 2005, pp. 117–124.

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

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