AN ABSTRACT OF THE THESIS OF

Brian D. King for the degree of Master of Science in

Electrical and Computer Engineering presented on June 12, 2012.

Title:

Adversarial Planning by Strategy Switching in a Real-Time Strategy Game

Abstract approved: _____

Alan P. Fern

We consider the problem of strategic adversarial planning in a Real-Time Strategy (RTS) game. Strategic adversarial planning is the generation of a network of high-level tasks to satisfy goals while anticipating an adversary's actions. In this thesis we describe an abstract state and action space used for planning in an RTS game, an algorithm for generating strategic plans, and a modular architecture for controllers that generate and execute plans. We describe in detail planners that evaluate plans by simulation and select a plan by Game Theoretic criteria. We describe the details of a low-level module of the hierarchy, the combat module. We examine a theoretical performance guarantee for policy switching in Markov Games, and show that policy switching agents can underperform fixed strategy agents. Finally, we present results for strategy switching planners playing against single strategy planners and the game engine's

scripted player. The results show that our strategy switching planners outperform single strategy planners in simulation and outperform the game engine's scripted AI.

Adversarial Planning by Strategy Switching in a Real-Time Strategy Game

by

Brian D. King

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 12, 2012
Commencement June 2013

Master of Science thesis of Brian D. King presented on June 12, 2012.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

---

Director of the School of Electrical and Computer Engineering

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Brian D. King, Author

ACKNOWLEDGEMENTS

I would like to thank my academic advisor Dr. Alan Fern for his support, broad knowledge, ideas, and persistent questioning. I would like to thank Dr. Michael Olsen for being on my defense committee. I would like to thank Dr. Thomas Dietterich, Dr. Prasad Tadepalli, and Dr. Weng-Keen Wong for being on my defense committee, and for their work creating a highly regarded Computer Science program for our beautiful state. I would like to thank my wife, Korakod Chimploy, for her support and patience, and for taking me by the hand and leading me to the admissions office when I mentioned that I might want to go back to school.

TABLE OF CONTENTS

Page

1 Introduction 1

2 Strategic Planning in Real-Time Strategy Games 4

3 Related Research 7

4 Architecture and Approach 10
- 4.1 Game Abstraction . . . 10
- 4.2 Strategies and Plans . . . 14
  - 4.2.1 Parameterized Strategies . . . 17
  - 4.2.2 Plan Generation . . . 18
- 4.3 Simulator . . . 20
- 4.4 Game System . . . 23
- 4.5 Combat Group Manager . . . 27
  - 4.5.1 Model . . . 27
  - 4.5.2 Integer Solutions . . . 29
  - 4.5.3 Learning . . . 31

5 Strategy Switching 33
- 5.1 Strategies in Markov Games . . . 33
- 5.2 Switching Theorem . . . 36
- 5.3 Monotone Maximin Strategy Switching . . . 39
- 5.4 Strategy Switching Planners . . . 41

6 Results 43
- 6.1 Experimental Setup . . . 43
  - 6.1.1 Strategy Set . . . 43
  - 6.1.2 Data Sets . . . 44
  - 6.1.3 Map Design . . . 46
- 6.2 Results . . . 48
  - 6.2.1 Maximin and Minimums . . . 48
  - 6.2.2 Simulation Accuracy . . . 55

TABLE OF CONTENTS (Continued)

Page

6.2.3 Switching Planner Choices 57
6.2.4 Switching Planner Performance 61

7 Summary and Future Work 62

Appendices 66
A Game State Grammar 67

Bibliography 67

LIST OF FIGURES

Figure		Page
2.1	Wargus Game Interface	5
4.1	Game Map	11
4.2	Abstract Game Map	11
4.3	A Strategic Plan	14
4.4	Plan Grammar	15
4.5	Example Plan Text	16
4.6	Make Plan	19
4.7	Get Compatibility	20
4.8	Simulator Application	23
4.9	Controller Architecture	24
4.10	Planning Cycle	25
4.11	Combat Linear Program	28
4.12	Assignment Graph	30
4.13	Node-Edge Incidence Matrix	30
5.1	Nash Equilibrium Linear Program	36
5.2	Monotone Minimax Selection	40
5.3	Switching Planner and Simulator	42
6.1	`2bases` Minimap	47
6.2	`2bases` Strategic Map	47
6.3	`the-right-strategy` Minimap	47
6.4	`the-right-strategy` Strategic Map	47
6.5	Scores in Simulation and Engine on `2bases`	56
6.6	Scores in Simulation and Engine on `the-right-strategy`	56

LIST OF TABLES

Table		Page
4.1	Properties of UnitGroup G	13
4.2	Strategy Template	18
4.3	Definition of Goal Orders	18
4.4	State Features	27
4.5	Higher-level Unit Types	28
4.6	Fixed Parameter C_j	32
4.7	Learned LP Parameters	32
5.1	Simple Game Matrix	34
5.2	Matching Pennies Game	35
5.3	Costs at State s_1	37
5.4	Costs at State s_2	37
5.5	Game Value at State s_1	38
5.6	Game Value at State s_2	38
5.7	Cost Sums for Action Sequence Pairs	38
5.8	Monotone Values at s_2 Based on Choices at s_1	40
5.9	Cost Sums for Action Sequence Pairs	40
6.1	Strategy Set 2012-02-05 Definition.	44
6.2	Stratagus Data Sets	45
6.3	Simulation Data Sets	46
6.4	Strategy Simulation Scores on `2bases`	49
6.5	Strategy Simulation Scores on `the-right-strategy`	49
6.6	Strategy Mean Scores on `2bases`	50
6.7	Strategy Mean Scores on `the-right-strategy`	50

LIST OF TABLES (Continued)

Table		Page
6.8	Strategy Win Rate on `2bases`	51
6.9	Strategy Win Rate on `the-right-strategy`	51
6.10	Switching Planner Scores in Simulation on `2bases`	52
6.11	Switching Planner Scores in Simulation on `the-right-strategy`	52
6.12	Switching Planner Mean Scores on `2bases`	53
6.13	Switching Planner Mean Scores on `the-right-strategy`	53
6.14	Fixed Strategy Maximin and Switching Planner Minimums in Simulation	54
6.15	Switching Planner Minimum Means in Engine	54
6.16	Strategy Pairs on `2bases`	55
6.17	Strategy Pairs on `the-right-strategy`	55
6.18	Strategy Choices of Switching Planners	57
6.19	`maximin` Choices by Epoch	58
6.20	`maximin` vs. `balanced 9` Choices in Simulator	59
6.21	`maximin` vs. `balanced 9` Choices	59
6.22	`monotone` vs. `balanced 9` Choices on `the-right-strategy` in Simulation	60
6.23	`monotone` vs. `balanced 9` Choices	60
6.24	Switching vs. Switching Win Rates on `2bases`	61
6.25	Switching vs. Switching Win Rates on `the-right-strategy`	61

LIST OF APPENDIX FIGURES

Figure Page

A.1 Game State Grammar 68

A.2 Game State Example 69

Chapter 1 – Introduction

Games are useful subjects of Artificial Intelligence (AI) research, because they can offer a high-level of intellectual challenge while having a well-defined structure. The well-known Turing Test [26] of intelligence was itself proposed as a game, and chess was a benchmark of AI progress [18] until computers began competing at an expert level [6]. Chess, checkers, and backgammon are games in which AI agents have won against expert human players [6], [21], [25]. These classic board games are turn-based, they have a small number of possible actions, the effects of actions are instantaneous and deterministic, and they have a small number of game objects with fixed attributes. In the restricted environment of these games, agents can search possible sequences of actions and select actions that lead to favorable states. However, in environments in which actions are continuous, durative, or stochastic in effect, and in which the number of agents and objects multiply, the search algorithms applied to board games do not scale. Also, the restricted dynamics and simple states of classic turn-based games are not much like complex real-world problems. To find new challenges for AI research, and to push for new tools to solve real-world problems, we need to test our agents in games that relax these restrictions.

Real-Time Strategy (RTS) games are simple military simulations that require players to build an economy, produce combat forces, and use those forces to defeat

their opponents. A defining feature of RTS games is that action proceeds continuously. A player can initiate actions at any time, rather than being restricted to turns, and the game continues whether a player takes action or not. An RTS player must solve problems of resource management, decision making under uncertainty, spatial and temporal reasoning, coordination, and adversarial real-time planning [5] in order to play succesfully. Actions have duration, and their effects are often randomized. Game objects have complex attributes that can vary over the course of the game due to damage or strengthening actions. In these ways RTS games are much more complex than classic board games, and come closer to representing real life scenarios. AI systems have been developed to play RTS games, but they still compete below the average level of human players in RTS competitions [28], [29], and so RTS games pose difficult and relevant challenges for AI researchers.

In this thesis we focus on strategic planning for the production and deployment of combat forces. In an RTS game, there are many strategies that a player can pursue. A player can try to build many cheap combat units and try to rush into battle, they can try to build a few powerful combat units to create more effective forces, they can deploy forces for defense or offense, they can concentrate or disperse their groups for attack, and a player can change strategies throughout a game. The problem of strategic planning is to define sequences of high-level actions that direct the production and deployment of groups of combat units and lead to defeating the opponent. One of the challenges of designing a strategic planning system is finding a level of abstraction that limits choices to a manageable number,

while providing enough detail to accurately model the dynamics of the game. The design of our strategic planning system is based on a strategic level abstraction of the game. In our game abstraction, territory is divided into regions, and units are clustered into groups. This greatly reduces the number of choices to consider. In our strategic plans, production actions specify the groups of units to produce and the mix of unit types within groups, and combat actions specify regions to attack or secure and what groups to deploy, rather than the paths or targets for individual combat units. Given an abstract game state, a planner generates task networks based on different strategies. Finally, we describe our game playing system. The architecture of the system has a configurable hierarchy of managers that execute a strategic plan.

The next section describes in detail the RTS scenarios we have choosen to solve.

Chapter 2 – Strategic Planning in Real-Time Strategy Games

The objective of an RTS game is usually to destroy all of the opponent's forces. In a typical scenario, a player begins a game with a single worker placed somewhere on a bounded territory called a "map". From the initial state the player must build an economy to train and support combat forces. The game used for this study is Wargus [3], a medieval fantasy combat game that runs on the Stratagus game engine [1]. The human player's view of the game is shown in figure 2.1. At the upper left corner there is an overview of the game called the "minimap". The minimap shows the player's forces in green and the opponent's forces in blue. The action bar on the left shows actions that can be assigned to the currently selected unit. The status bar at the top shows the resources held by the player. The center right shows a movable window on the game map. The map view shown has been edited to label some of the game units. This player has several footmen, a town hall, a barracks, and farms. Gold mines are not owned by players, but peasants can mine them for gold.

In Wargus, the economy consists of peasant workers, gold, timber, and oil resources, and the production from different types of buildings. Peasants mine gold and harvest timber. Gold can be used to recruit and train new workers. Gold and timber can be used to construct buildings. Some buildings are used to train combat units such as footmen and archers. Other buildings, such as a blacksmiths,

enhance the attack and defense strength of combat units. After a combat unit has been trained, it can be sent on patrol, to a location, or to attack another unit.

The strategic planning problem that we address in this thesis is planning for the training and deployment of combat units. We do not address resource production and build order, so in the scenarios we test each player starts with enough resources and buildings to train and support a large combat force. Given that a player has the resources to train different types of combat units, a plan has to specify which building groups are used to train combat units, how many of each type to train,

Figure 2.1: Wargus Game Interface

how to organize them into groups, where to deploy them, how they should attack or defend, and in what order these tasks should be executed. In section 4.2.2 we describe the planning language we use for strategic planning in the Wargus RTS game.

Chapter 3 – Related Research

Commercial combat simulations have been used as AI testbeds since at least 2001 [13], and since 2003 RTS games have been used because of their challenging demands [5] on decision making. Commercial combat simulators often include an AI player in order to provide an opponent when playing a game alone. So there has been at least a decade of research in this area. Most RTS AI controllers are of two types: they either use scripting or simulation. This section gives an overview of these two approaches to RTS AI, and we compare and contrast our own AI controller to these approaches.

Most RTS game playing controllers operate by some form of scripting. At its simplest, an engineer codes a controller to recognize game states and trigger a related script that executes some human engineered tactic. Sophisticated versions of scripting add the ability to learn or reason about which scripts should be executed in which states. Examples of scripting with learning for game playing AI include Case-Based Reasoning [27] and Dynamic Scripting [24]. Examples of scripting with reasoning are Goal-Driven Autonomy (GDA) [28], and tactic selection by symbolic reasoning [30]. The disadvantages of these techniques are that the AI has a limited number of scripts to choose from and the composition of the scripts is a labor intensive and error-prone process. If we define "understanding" as the ability to predict the outcomes of events, then scripted controllers have little to no under-

standing of the domain that they act in. In contrast, our controller uses simulation to predict outcomes, indicating that it has some understanding of its domain. Our controller generates plans from parameterized strategies, so the number of possible plans is large, and in principle new strategies could be generated by automatically varying the strategy parameters. Though low-level modules could be implemented by scripting, as we show with our combat controller 4.5, low-level modules can also be trained, thereby avoiding a disadvantage of manual script creation.

Another class of controllers use game simulation to look ahead and evaluate alternative actions. In Monte Carlo Planning [10], the AI player randomly generates high-level plans for all players, simulates the plans as many times as possible, and then executes the plan that gives the best statistical result. But defining the best result may not be straightforward. From Game Theory [15], we know that the highest value result among pairs of strategy choices may not be the best choice in an adversarial game, because an opponent who can predict a player's strategy choice may be able to exploit a weakness in that strategy. Choosing a strategy by a Maximin or Nash equilibrium evaluation after plan simulation is a refinement of Monte Carlo planning that has been used in RTS games. In the RTS planner described by Sailer et. al. [20], the actions that are evaluated are actually high-level strategies that are simulated to completion. However, the implementation of these strategies is not described, so we do not know how they are defined, how they are translated into unit actions, or how these strategies direct resource production. In this study, we define an RTS strategy as a set of parameters, and we present an algorithm that generates a plan as a high-level task network. An advantage of

parameterized strategies is that it is clear what the strategy space is. By having a high-level task network, it is clear how to incorporate production tasks into the simulation and execution of a plan, and the high-level plan suggests how to assign tasks to controllers. An additional feature of our architecture is that our simulator uses a spatial abstraction that includes strategic paths, so path lengths in the terrain are considerations when evaluating strategies, and our simulator includes production time requirements.

Chapter 4 – Architecture and Approach

In this section, we describe the abstraction we use for simulation, the plan generation algorithm, our hierarchical architecture, the simulator used to evaluate strategy pairs, and the strategy switching algorithm which is our focus. In addition, we describe a low-level module, the combat controller, to show how different modules can be plugged into the architecture to create a game player.

4.1 Game Abstraction

A Wargus game map is a grid of cells, typically 32x32 or 64x64. A common action for combat units is the "MoveAttack", which means to move to a cell of the map and attack whatever is there. If we take a simple combat scenario of 10 footman on a 32x32 map, there are approximately 10^{30} possible permutations of MoveAttack commands that a player could issue to their units. Contrast this with chess, which has an average 35 possible moves at each turn, and we see that deciding the actions for just one Wargus update is a daunting challenge. Of course, it does not take much game experience to see that many of these moves are equivalent, and that the strategic difference between moves to nearby cells is negligible, so we can make great reductions to the size of the action space by clustering map cells in a spatial abstraction.

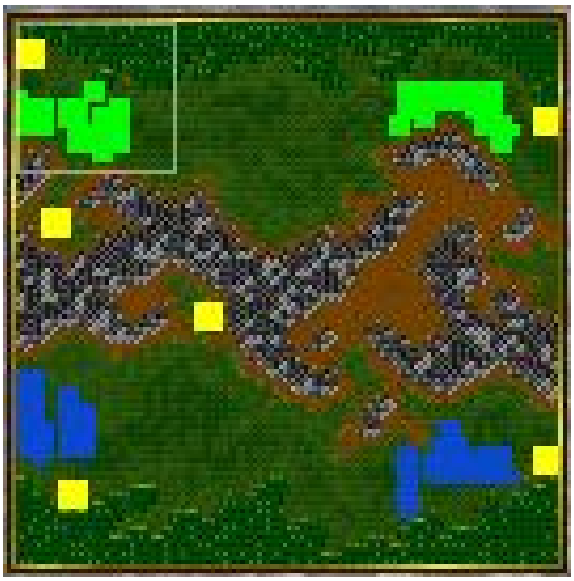

Figure 4.1: Game Map

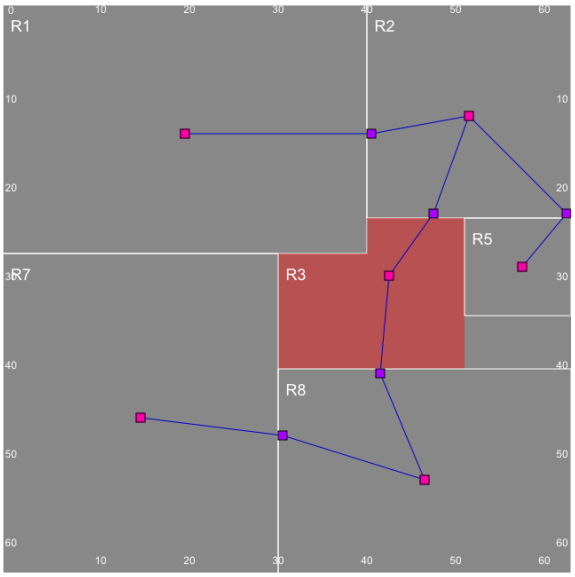

Figure 4.2: Abstract Game Map

Figure 4.1 shows a map called `one-way-in-one-way-out` that we have chosen for some of our experiments. This figure is a screenshot of the "minimap" overview of the entire map. For the experiments, we start with two groups of production units, called "bases", for the player and the opponent. Green squares show the player's units, blue squares are the opponent's units, and gold squares are gold mines. This map configuration is called "2bases" in our experiments. Figure 4.2 shows a hand-coded abstraction of this map into 8 regions labeled $R1$ through $R8$ ($R4$ and $R6$ are small, and their labels do not show in the figure). The figure shows the connectivity graph, and a region containing a chokepoint in red. A chokepoint is a narrowing of a region that forces units to present a smaller front as they pass through.

A map abstraction consists of regions and a connectivity graph showing ab-

stract paths between regions. Regions can be marked as chokepoints to show where passing units will be vulnerable and where units can prevent passage of an opponent's forces. Since the finest divisions of the map are square cells, we abstract a region as a collection of contiguous rectangles, which are sufficient to create any desired partition of a map. For games that use continuous coordinates, it might be more appropriate to use a triangular mesh to allow us to approximate any shape, but in Wargus shapes other than rectangular could create ambiguity about which region border cells belong to, so there is no advantage in using more flexible regions. The connectivity graph shows connections between a region's center and points on the border between regions. Each connection has a length which is intended to quantify the difficulty of moving from region to region.

With this spatial abstraction, we greatly reduce the number of "MoveAttack" actions that have to be considered. For our particular map, a unit starting in region one can move to only seven other regions, and there is a unique path to each region. We designed this abstract map so that these choices represent what is strategically important - the choice of moving to occupy a chokepoint, moving to defend allied units, or moving to attack enemy units. Tactical decisions can be made in the game map after units have reached a strategic target.

Combat units have been organized into hierarchies since ancient times, and reasoning about combat units as groups is part of the earliest formalizations of military analysis [14], [16]. There are many approaches to modeling combat groups. The SORTS system [30] motivates grouping by appealing to Gestalt theory. There are different methods of aggregating unit speed, survivability, and attack strength.

In our abstraction, units are classified into two types: combat and production. Combat units are such things as footmen, archers, knights, and machines like ballistas. Production units are peasants, town halls, barracks, and anything else that produces other units. Given a state in an ongoing game, all units of the same player, class, and region are grouped into an object called a `UnitGroup`.

Properties of `UnitGroup` G are calculated from the properties of the units g assigned to them. The properties that are sums, minimums, or maximums of the constituent unit properties are given in table 4.1.

Property	Formula
$MaxSpeed(G)$	$= \min_{g \in G} MaxSpeed(g)$
$Armor(G)$	$= \sum_{g \in G} Armor(g)$
$BasicDamage(G)$	$= \sum_{g \in G} BasicDamage(g)$
$PiercingDamage(G)$	$= \sum_{g \in G} PiercingDamage(g)$
$MinAttackRange(G)$	$= \min_{g \in G} MinAttackRange(g)$
$MaxAttackRange(G)$	$= \max_{g \in G} MaxAttackRange(g)$

Table 4.1: Properties of UnitGroup G

Given a game state, we can create an abstract game state as described above that has all the features needed to make strategic decisions. We have created a grammar for abstract game states, so that the strategic concepts are clearly defined, and states can be saved and analyzed. The grammar for the abstract game state and an example state are given in appendix A. The last abstraction we need for taming complexity is a representation of the actions of groups of units. We define high-level tasks to represent group actions in the next section.

4.2 Strategies and Plans

> No battle plan survives contact
> with the enemy.
>
> Helmuth von Moltke

Tasks are high-level actions assigned to unit groups. We implemented three task types for our system: produce, secure, and attack. A "produce" task can be assigned to a production group, and "secure" and "attack" tasks can be assigned to combat groups. In our system, a plan is a directed graph of tasks and a set of unit groups that execute those tasks. The graph nodes are tasks and edges are "triggers". Tasks can be started when prior tasks connected by a trigger have ended or started, and these conditions are indicated by the type of the trigger. An example plan is shown in figure 4.3.

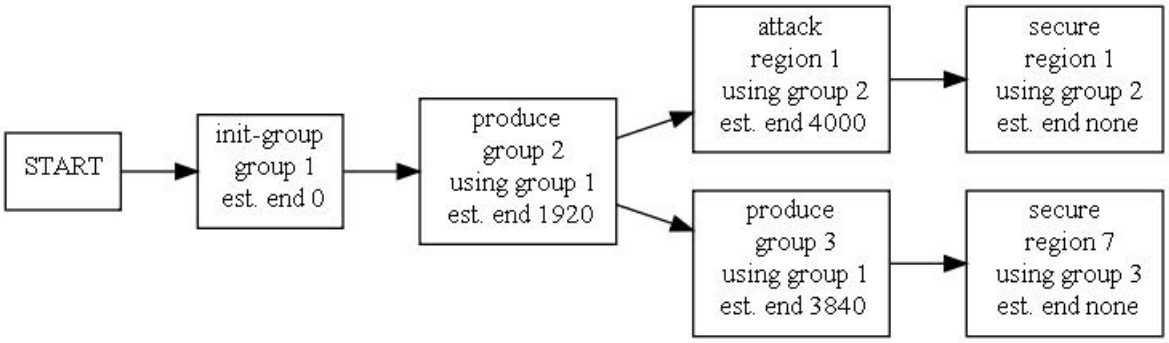

Figure 4.3: A Strategic Plan

We describe plans using a planning language. A planning language constrains the number of possible plans, and having a language means we can save plans

```
plan           : '(' ':plan' NAME ':player' INT group_spec* task* ')' ;
group_spec     : '(' ':group-spec' INT ':type' NAME units_spec*
                 (':initial-units' '(' INT* ')')? ')';
units_spec     : NAME INT;
task           : '(' ':task' NAME task_args ':type' NAME (':using' INT)?
                   (':start' start_triggers)?
                   (':end'   end_triggers)? ')' ;
task_args      : '(' group_arg? region_arg? ')';
group_arg      : '(' ':group' INT ')' ;
task_arg       : ((NAME INT)|NAME);
region_arg     : '(' ':region' INT ')';
start_triggers : '(' ':trigger' trigger* ')' ;
end_triggers   : '(' ':trigger' trigger* ')' ;
trigger        : '(' ('start' | 'end') NAME ')' ;
```

Figure 4.4: Plan Grammar

for analysis and testing. Figure 4.4 shows the grammar of our planning language given in ANTLR [19] format. `group_spec` tells the composition of groups that will execute the plan. `tasks` have a type, a region or group identifier given in the `task_args`, and a `:using` property that tells which group to use to execute the task. If the task is a combat task, then the argument is a region identifier. If the task is a production task, then the argument is the identifier of a group to produce. The text corresponding to the plan in figure 4.3 is shown in figure 4.5.

The types of tasks are not restricted by the grammar, but we defined `produce`, `attack`, and `secure` types. The `:using` group of a production task will be a group of buildings or workers (peasants in Wargus) that can be used to produce other buildings, workers, or combat units. The argument to a `produce` task is the group identifier referring to the group to produce. An `attack` task directs a combat

```
(:plan plan_0 :player 0
  (:group-spec 1 :type group-building
    unit-town-hall 1 unit-elven-lumber-mill 1 unit-human-barracks 1
    unit-farm 7 unit-peasant 1)
  (:group-spec 2 :type group-combat unit-archer 2 unit-footman 3)
  (:group-spec 3 :type group-combat unit-archer 2 unit-footman 3)

  (:task init-group1 ((:group 1)) :type init-group
    :end (:trigger (start produce1))
  )
  (:task produce1 ((:group 2)) :type produce :using 1
    :end (:trigger (start attack2)(start produce4))
  )
  (:task attack2 ((:region 1)) :type attack :using 2)
  (:task secure3 ((:region 1)) :type secure :using 2)
  (:task produce4 ((:group 3)) :type produce :using 1
    :end (:trigger (start secure5))
  )
  (:task secure5 ((:region 7)) :type secure :using 4)
)
```

Figure 4.5: Example Plan Text

group to go to a region and attack whatever enemy units are there. If enemies are eliminated, the `:using` group is available for the next task. The `secure` task is the same as the `attack` task except that it does not end. After enemies are eliminated the securing group remains in place. The task can be ended explicitly by an "end" trigger.

In a strategic combat game, the highest-level goals are to control territory. Control of a region means that a player has the freedom to move about and use the resources of that region without significant risk of attack from an opponent. At the end of the game, one player will control all the regions of the map by

having eliminated the enemy. In intermediate stages, a player can gain control of a region and begin using the resources of that region. So a strategic plan consists of sequences of production and combat tasks that extend a player's control to all regions of a map.

4.2.1 Parameterized Strategies

There are many tradeoffs to consider when creating a strategic plan, such as what type of combat units to train and when and where to send them to battle. Players organize different approaches to these tradeoffs into strategies. A typical RTS strategy is a "rush", in which a player tries to train a small number of combat units quickly and send them to attack the enemy before the enemy has time to build adequate defenses. A contrasting strategy is "turtling", in which a player tries to build a large defensive force to survive an initial attack. We have organized strategies as sets of parameters called strategy templates. The strategy templates encode strategic tradeoffs. A plan generation algorithm can then create a plan when given a strategy template and a game state.

The first three parameters tell what size group to produce for different combat goals. The planner recognizes three types of goals: secure a base, secure an enemy base, and secure a chokepoint. A strategy that emphasizes defense will define larger groups to secure a player's bases. Another factor distinguishing offensive from defensive strategies is the order in which goals are pursued. The "goal order" parameter is an enumeration of different goal priorities, as given in table 4.3. For

Parameter	Range	Definition
base force	{1,...,9}	size of groups that defend bases
enemy base force	{1,...,9}	size of groups that attack enemy bases
chokepoint force	{1,...,5}	size of groups that secure chokepoints
goal order	{0,...,5}	priority of bases, enemy bases, and chokepoints
MassAttack	{false,true}	groups attack jointly or separately
time to target	[0,1]	weight of time to target factor for goal assignment
enemy damage	[-1,1]	weight of damage that enemy in goal region can deliver
damage ratio	[-1,1]	weight of ratio of allied to enemy damage

Table 4.2: Strategy Template

ID	Name	Priority Order
0	defensive	allied,chokepoint,enemy
1	defend-attack	allied,enemy
2	attack-defend	enemy,allied
3	chokepoint	chokepoint,allied,enemy
4	offensive	enemy,chokepoint,allied
5	offensive only	enemy

Table 4.3: Definition of Goal Orders

example, if a plan is being generated for a strategy with a "defensive" goal order, then the planner will prioritize production of combat groups to secure allied bases, then chokepoints, and then enemy bases.

The use of the *MassAttack*, *time to target*, *enemy damage*, *damage ratio* parameters is discussed in 4.2.2.

4.2.2 Plan Generation

Our base plan generation algorithm is in the class `GoalDrivenPlanner`. To make a plan, `GoalDrivenPlanner` is given a strategy template, a set of group definitions,

```
makePlan(strategy, state, groups)
 1  plan ← initialize plan with groups
 2  goals ← goals from state
 3  sort goals according to strategy's goal order
 4  for goal in goals
 5        do task, group ← AssignGroup(plan, goal)
 6           if group is not null
 7              then addCombatTask(plan, task, group)
 8              else  group ← defineGroup(goal)
 9                    addCombatTask(plan, task, group)
10  if MassAttack(strategy)
11     then patch plan to combine attacks on enemy bases.
12  return plan
```

Figure 4.6: Make Plan

and the current state. The planning function of the `GoalDrivenPlanner` creates a set of goals for the current state. It then attempts to satisfy these goals in priority order by assigning an available combat group to each goal. If no combat group is available, the planner defines a potential combat group, and tries to find a production group that can produce the combat group. The high-level algorithm *makePlan*() is given in figure 4.6.

The *AssignGroup*() function finds available combat groups in the current plan and determines which group is most compatible with the given goal by calling *GetCompatibility*() 4.7. Strategy parameters *time to target*, *enemy damage*, and *damage ratio* are passed to the *GetCompatibility*() function as weights w. The *addCombatTask*() function adds a combat task to the end of the plan if the group already exists. If the group does not exist, the function looks for a free production

group and adds a sequence of tasks to the plan to produce the combat group and send it to secure the goal region.

$GetCompatibility(w, group, goal)$
1 $t \leftarrow time\_to\_target(group, goal)$
2 $s \leftarrow enemy\_damage(goal)$
3 $r \leftarrow damage\_ratio(group, goal)$
4 **return** $w_t t + w_s s + w_r r \triangleright$ (high value is more compatible)

Figure 4.7: Get Compatibility

4.3 Simulator

To evaluate strategic plans, we need to approximate the outcome of the sequences of actions defined by the plan, which we can do by simulating these actions in the abstract game state described in 4.1. So we need to estimate the outcomes of our `produce`, `attack`, and `secure` actions defined in 4.2.

The goal of a `produce` task is to create a new `UnitGroup` with a specified number of units in it. The Wargus configuration files specify the prerequisites and time needed to produce a unit. The `produce` task is implemented in the simulator by an object that verifies that the game state has the prerequisite resources and units, and tracks the completion progress. To estimate the time to completion, the action loads a predefined graph of unit dependences along with the time need to complete a unit. When the game cycle advances past the number of cycles needed to create a unit, the action object adds a new `UnitGroup` to the abstract state or

updates the `UnitGroup` attributes to indicate that it has the hitpoints, damage potential, armor, etc. of an additional unit. For example, training a footman requires a barracks and takes 360 game cycles. Given a goal to produce 10 footmen, the action object will verify that there is a barracks available. After 350 game cycles have passed, it will create a `UnitGroup` with the attributes of one footmen. After each additional 360 cycles, the `UnitGroup` will be updated with the attributes of another footman.

`attack` and `secure` are our two types of combat tasks. The goal of these tasks is to secure a region by destroying all opponent units in the region. An `attack` task completes when the opponent's units are destroyed, while the `secure` never finishes because it is an ongoing task of occupying a region. Combat tasks are executed by a combat object in the simulator. Combat simulation works by moving `UnitGroups` along paths in the abstract map's connectivity graph until an opponent group is encountered or the group reaches the target region. When `UnitGroups` are in the same region, we assume that they can attack each other, and as soon as a `UnitGroup` being used by a combat task meets an opponent group, it will attack it. Combat proceeds by calculating damage points and subtracting damage from the hitpoints of each group. Damage for `UnitGroups` in the simulator is calculated the same way as it is for units in the engine. Damage to an opponent inflicted by an ally is calculated as

$$damage = ally.getBasicDamage() - opponent.getArmor() \tag{4.1}$$

When `UnitGroups` are in combat, they stop motion and attack until one group is destroyed (hitpoints drop to zero). The winning `UnitGroup` can proceed to its target region.

A simulation is updated in a time increment. The game state is set forward by the increment amount, then all the active actions are executed. Actions calculate a set of attribute differences between the previous cycle and the new cycle. Attribute differences are `UnitGroup` hitpoint changes and position changes along the edges of the map connectivity graph. The attribute differences are collected from all active actions, and then applied to update the game state. The purpose for delaying attribute update is to simulate simultaneous actions. Without the delay, the order of action execution would be significant. For example, a group might cause enough damage to destroy an opponent group before the opponent had a chance to attack, while in the engine the individual units of the opponent group might have many opportunities to attack before the whole group was destroyed. It is necessary to collect the attribute changes and apply them after all groups have had a chance to act in an update.

Though visualization of the simulation is not needed for planning, it is useful for debugging. The simulator application shown in figure 4.8 was used to debug the simulator.

The details of how the simulator is used for game value estimation are described in chapter 5.

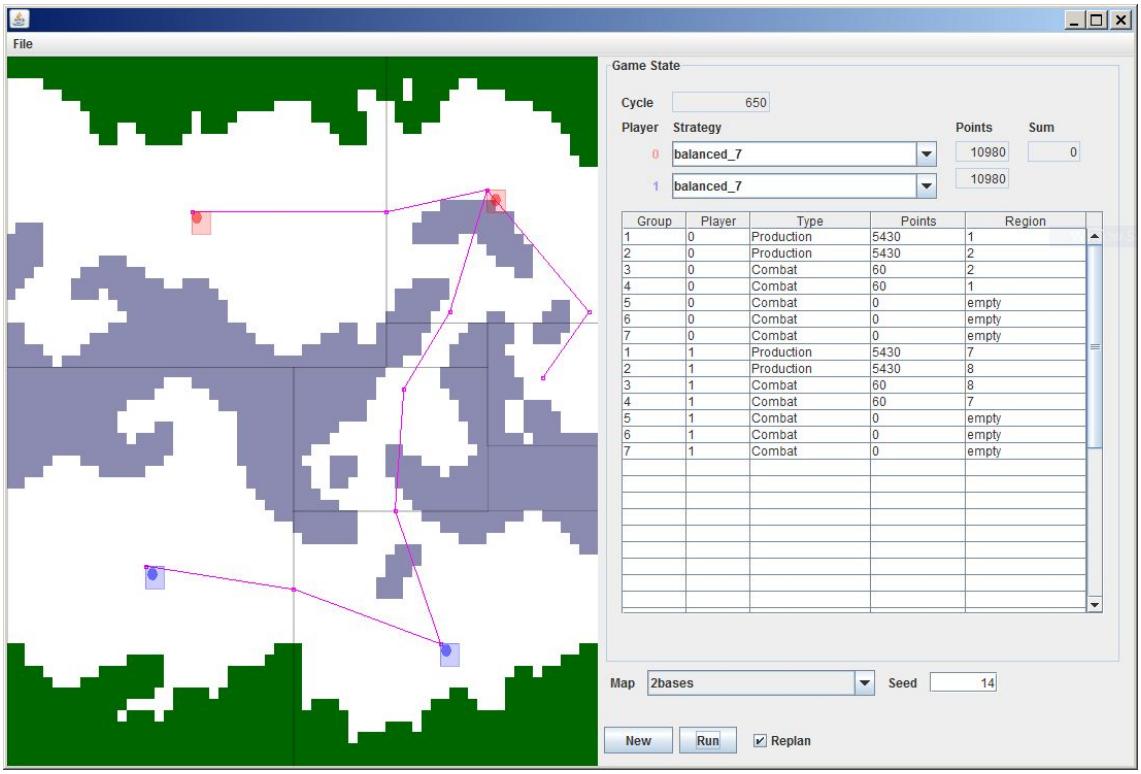

Figure 4.8: Simulator Application

4.4 Game System

The game playing system consists of a client application and a Stratagus engine, which communicate through an internet socket. The client application runs one or more controllers, each of which controls the game units for one player. The controllers are run by a class called the `GameRunner`. Controllers are implemented by the `StrategyController` class. Controllers are configurable, but for our experiments they each have a planner and a hierarchy of managers. The system structure is shown in figure 4.9. The hierarchy of managers structure is a common

approach to multi-agent systems [12], and maps naturally to military command structures. The approach was inspired by the hierarchy of managers used by McCoy and Mateas [17] for playing Wargus, though our implementation uses only two sub-managers.

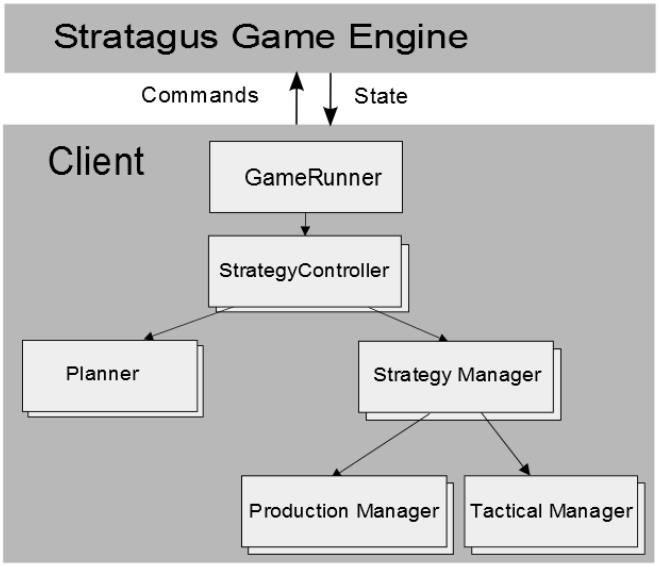

Figure 4.9: Controller Architecture

The `GameRunner` coordinates the interaction between the Stratagus engine and the player's controllers. At the beginning of a game (also called an episode), the `GameRunner` passes the initial state to the `StrategyController`s. The controllers may create a plan and return unit commands to the runner, and the runner then tells the engine to execute a fixed number of game cycles, and passes it any unit commands it has received. After the specified game cycles, the runner receives the updated state and passes it to the controllers. The controllers may replan when

they are updated. This interaction forms the planning cycle shown in figure 4.10.

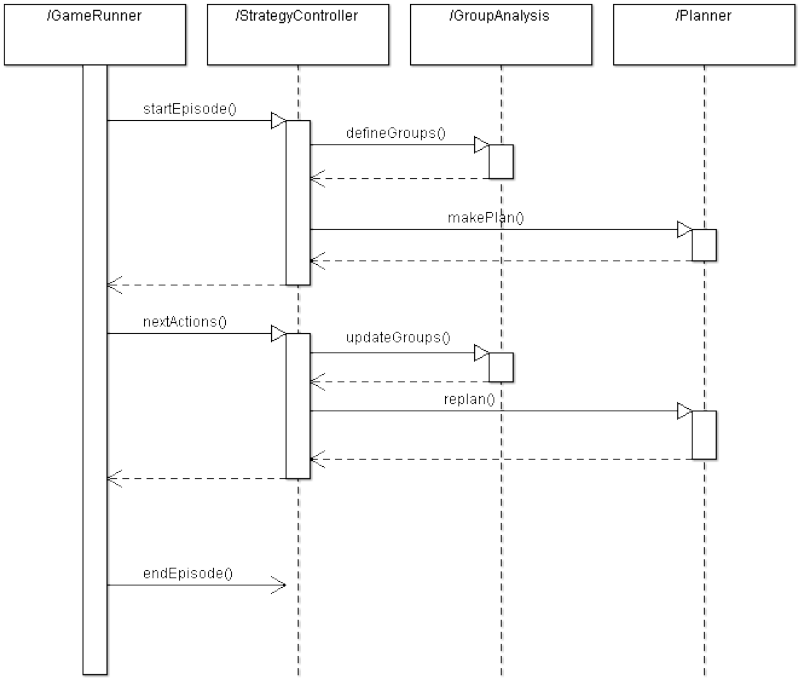

Figure 4.10: Planning Cycle

The managers under the control of a `StrategyController` are responsible for executing a plan. The `StrategyManager`, `ProductionManager`, and `TacticalManager` are all sub-classes of an abstract `Manager` class. The `Manager` class defines the interface for a task manager that performs a given task. It has several methods that allow it to communicate state and task status with both its parent and child Managers. The `StrategyManager` assigns tasks from a plan to its sub-managers,

and tracks which tasks are active. When an updated game state is given to the `StrategyManager`, it passes the state on to the sub-managers. If they detect that their task is complete, they signal back to the `StrategyManager` which marks the task as complete. When all the predecessors of an inactive task are complete, then that task is marked as active and can be assigned to a sub-manager.

Groups may lose units when they are attacked, and re-planning may define new groups, so as the game progresses, a player could be managing a large number of small groups. There is no peer-level messaging in the manager hierarchy, so groups in the same region can not be coordinated and may interfere with each other. To prevent this, the `StrategicController` joins combat groups that are in one region into a new group before re-planning. Joining and defining groups is done by the `GroupAnalysis` class.

The `ProductionManager` is responsible for creating unit commands to execute production tasks. A strategic plan may contain a high-level task such as "produce a group of 5 footmen using production group 1". The `ProductionManager` will find a barracks in production group one and issue the commands needed for the barracks to train the required footmen. The `ProductionManager` tracks events in the state update and signals back to its parent manager when the task is complete.

The `TacticalManager` controls combat groups. Given a task to attack or secure a region, it will create its own sub-manager, an instance of the `CombatGroupManager` for the task that controls a combat group. The implementation of the `CombatGroupManager` used for our experiments is described in section 4.5.

4.5 Combat Group Manager

The game system can be configured with any implementations of the `Manager` interface. In this section we describe an implementation that executes a combat task, to give an example of a unit group manager.

4.5.1 Model

When securing an opponent region, a combat group must accomplish two tasks: killing all opponent combat units, and disrupting the opponent's production of new combat units. To accomplish both tasks, the attacking units must trade off attacks on existing combat units, buildings, and peasants. If we express the value of assigning an allied combat unit to attack an opponent unit as a linear combination of state features, then these tradeoffs can be expressed in a linear programming model.

We write the combat group's objective as a parameterized function of state features and unit actions. The state features are given in table 4.4.

$K_{j,k}$	indicator that opponent j is unit class k
$p_{i,j}$	proximity of ally i and opponent j
t_j	opponent j can attack allied combat group

Table 4.4: State Features

To limit the number of parameters $K_{j,k}$ for unit types, we grouped the types into a higher level of five classes, which are given in table 4.5.

Let $x_{i,j}$ be the action that ally unit i attacks opponent unit j, and let there be

N allied units and M opponent units. Then objective function $\hat{Q}_\theta$ is

$$\hat{Q}_\theta(x,p,t,K) = \sum_{i,j}^{N,M}(\theta_1 p_{i,j} + \theta_2 t_j + \theta_3 K_{i,1} + ... + \theta_{2+k} K_{i,k})x_{i,j}$$

Let opponent capacity C_j be the maximum number of allied units that can effectively be assigned to opponent j (values are in table 4.6). The resulting linear program (LP) is shown in figure 4.11.

$$\begin{array}{lll} \text{Maximize} & \hat{Q}_\theta(x,p,t,K) & \\ \text{subject to} & \sum_j^M x_{i,j} \leq 1 & \text{for } i = 1...N \\ & \sum_i^N x_{i,j} \leq C_j & \text{for } j = 1...M \\ & x \geq 0 & \end{array}$$

Figure 4.11: Combat Linear Program

A solution to this LP is a vector x that maximizes the value of a combat group's attacks. In the implementation, LP 4.11 is solved using the Gnu Linear Programming Kit (GLPK). In the next section we show how to interpret the solution.

Class	Index
Peasant	1
Combat	2
Combat Bldg.	3
Production Bldg.	4
Support Bldg.	5

Table 4.5: Higher-level Unit Types

4.5.2 Integer Solutions

LP 4.11 is a variation of the Assignment Problem [23, 7.1.3(2)] in which there can be a different number of units in the two sets. The value of $x_{i,j}$ represents the degree to which ally i attacks opponent j. $x_{i,j}$ is restricted to the range $[0, 1]$ by the constraints, but we cannot assign an attack fractionally, so we need to know that an optimal solution will be integer. The LP has the form $\max\{cx \mid Ax \leq b, x \geq 0\}$. It has been shown that LP problems of this form have integer optimal solution if constraint matrix A is *totally unimodular* and b is integer [23, Theorem 7.2]. Our bounds vector b is integer because we have defined C_j to be integer, so we just need to show that our constraint matrix is totally unimodular.

This assignment problem can be represented as an undirected bipartite graph in which allied units form one set of nodes, opponent units form the other set, and edges are the possible target assignments. Our constraint matrix A is the *node-edge incidence matrix* corresponding to this graph. The node-edge incidence matrix is the (0,1)-matrix in which row i corresponds to node i and column i, j to edge i, j. If i, j is an edge of the graph, $A_{i,(i,j)} = A_{j,(i,j)} = 1$, otherwise the entries of A are zero. An example graph for 2 allied units u_i and 3 opponent units t_j, and the corresponding node-edge incidence matrix are shown in figures 4.12 and 4.13. We can show that the constraint matrix for problem 4.11 is totally unimodular using the following theorem:

Theorem (Sierksma [23, 7.3]) Sufficient condition for total unimodularity. *Any (-1,0,1)-matrix A is totally unimodular if*

(1) each column of A contains not more than two nonzero entries, and

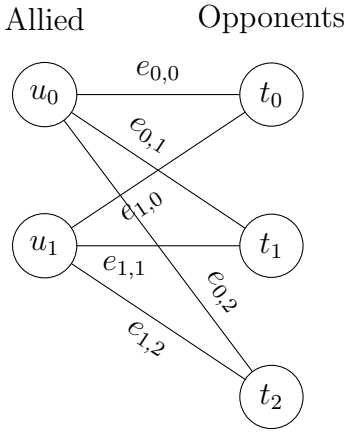

Figure 4.12: Assignment Graph

$$\mathbf{A} = \begin{array}{c} \\ u_0 \\ u_1 \\ \hline t_0 \\ t_1 \\ t_2 \end{array} \begin{array}{c} \begin{array}{cccccc} e_{0,0} & e_{0,1} & e_{0,2} & e_{1,0} & e_{1,1} & e_{1,2} \end{array} \\ \left(\begin{array}{cccccc} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right) \end{array}$$

Figure 4.13: Node-Edge Incidence Matrix

(2) the rows of A can be partitioned into two subsets such that:

- *(i) if a column contains two entries with the same signs, then the corresponding rows belong to different subsets, and*
- *(ii) if a column contains two entries with the opposite signs, then the corresponding rows belong to the same subset*

The rows of constraint matrix A of problem 4.11 can be partitioned into two sets. The first set, shown in the upper half of figure 4.13, corresponds to constraints $\sum_j^M x_{i,j} \leq 1$ for $i = 1...N$. The second, shown in the lower half of the figure, corresponds to constraints $\sum_i^N x_{i,j} \leq C_j$ for $j = 1...M$. In the first set, $A_{i,(i,j)} = 1$, in the second set, $A_{j,(i,j)} = 1$, and all other entries are zero. Each column (i, j) has exactly one nonzero entry from the first set, and exactly one from the second set, so condition (1) is satisfied. The nonzero entries of each column have the same sign and are from different row subsets, so condition (2) is satisfied. A is totally unimodular, so an optimal solution to LP 4.11 gives an integer assignment $x_{i,j} \in \{0, 1\}$. A value of 1 means that ally i should attack opponent j, and the

LP constraints assure that an ally will be assigned to at most one opponent, so an optimal solution x is a valid multi-agent attack assignment.

4.5.3 Learning

There are many parameters to set in the model, so we would like the controller to be able to learn these itself. The objective function of the LP is a parameterized action-state value function (Q-function) with state features and multi-agent actions $x_{i,j}$. This suggests that Q-learning could be used to learn the parameters. Unfortunately, when using gradient ascent to update parameters, they tended to increase without converging, possibly because of the instability of the max function in the Q-learning update rule, so this approach was abandoned.

Instead we used coordinate ascent to learn the parameters. Coordinate ascent found a stable set of parameters that produced a successful controller. In coordinate ascent, a range for each parameter is fixed, and each parameter is incremented through its range in turn, and the parameter value that produces the highest reward is kept. The learned parameters are given in table 4.7. The combat controller using these parameters won consistently over the Stratagus built-in script and an earlier controller trained using OLPOMDP in tactical combat scenarios.

The learned parameters show that proximity and whether an opponent unit is able to attack a unit of the combat group are the most important factors for deciding which unit to attack. Unsurprisingly, the second most important factor is whether or not the opponent unit is a combat class unit (class 2). The third most

important factor is whether the opponent unit is a peasant or a production building. The prioritization of attacks on peasants is a difficult decision, because of the many roles they play in the game, and because of their unpredictable movement.

It is worth noting that the earlier OLPOMDP controller was successful in tactical scenarios against combat units only, but it was unable to learn the tradeoffs needed to pursue the dual tasks of killing existing units and disrupting opponent production.

Unit Type	C_j
FOOTMAN	3
PEASANT	2
BALLISTA	3
KNIGHT	3
ARCHER	3
FARM	3
BARRACKS	6
STABLES	6
LUMBER MILL	6
FOUNDRY	6
TOWN HALL	8
MAGE TOWER	4
BLACKSMITH	4

Table 4.6: Fixed Parameter C_j

Param.	Value	Feature Description
θ_1	0.5	Proximity
θ_2	0.5	Opponent unit can attack
θ_3	0.3	Opponent unit is class 1
θ_4	0.4	Opponent unit is class 2
θ_5	0.1	Opponent unit is class 3
θ_6	0.3	Opponent unit is class 4
θ_7	0.1	Opponent unit is class 5

Table 4.7: Learned LP Parameters

Chapter 5 – Strategy Switching

5.1 Strategies in Markov Games

In section 4.2.1 we defined a parameterized strategy template. Using the template, it is a simple matter to generate sets of strategies. In Markov Decision Processes (MDPs) it has been shown that an agent with a set of policies can perform at least as well as any single policy by switching among policies in the set [9]. Sequential decision problems such as our Wargus game can be formalized as MDPs, however MDPs have only one decision making agent, and we have to consider other agents as random influences from the environment. Markov Games [11] (also called Stochastic Games) extend MDPs to multi-agent decision problems by incorporating solution concepts from Game Theory.

The concepts that we need from Game Theory to begin understanding multi-agent policy switching are the game matrix, the maximin (or minimax) strategy, and Nash Equilibrium, where a "strategy" in Game Theory terminology corresponds to an action in an MDP. Table 5.1 is an example of a game matrix. This matrix represents a game in which the score is decided by the joint actions of a player and an opponent. The scores are the values awarded to the row player, and the negation of these scores are awarded to the opponent. Since the players' scores sum to zero, this is called a zero-sum game. The player attempts to maximize

V	ϕ_1	ϕ_2	min	maximin
π_1	1	-1	-1	
π_2	0	0	0	*
max	1	0		
minimax		*		

Table 5.1: Simple Game Matrix

the score when choosing a row action, and the opponent attempts to minimize the player's score (and maximize their own) when choosing a column action. Knowing the values, the player could be tempted to select action π_1 to receive the winning score of 1. But the opponent also knows the outcomes, and so would take action ϕ_2, resulting in the player receiving -1 and losing. The safe option for the player is to choose action π_2 and settle for a tie.

In a zero-sum game in which both players are rational and have perfect information, a player has to assume that the best they can expect to do is to maximize their worst-case. This can be done by choosing the action that returns the maximum of the minimum values of the possible choices. The value returned is called the maximin value (or security level), and the action that guarantees it is called a maximin strategy (also called "maxmin" or "security strategy" [15]). For game matrix $V(\pi_i, \phi_j)$, the maximin strategy is given by

$$\arg\max_{\pi_i} \min_{\phi_j} V(\pi_i, \phi_j). \tag{5.1}$$

In the game in table 5.1, both players have a security level of zero given by player action π_2 and opponent action ϕ_2. The maximin-minimax actions are marked by

V	ϕ_1	ϕ_2	min	maximin
π_1	1	-1	-1	*
π_2	-1	1	-1	*
max	1	1		
minimax	*	*		

Table 5.2: Matching Pennies Game

a "*". No player can improve their security levels by changing their action, so the action pair is said to be an equilibrium.

When security levels of two players differ, the players can improve their expected values by randomizing their choices. This is called a "mixed strategy" in Game Theory, or a stochastic policy in an MDP. If we change the scores from table 5.1 to those in table 5.2, we get the Matching Pennies game. In this game the row player wins when their action matches the opponent's. Both actions are security strategies for both players, the player's maximin value is -1, and the opponent's minimax value is 1. In the Matching Pennies game, the optimal strategy is to choose either action with 50% probability, which improves the security level for both players to zero. There is always an optimal probability distribution over actions in a two-person, zero-sum game of finite actions, which is known as the Nash Equilibrium [15]. Further, computing the Nash Equilibrium can be formulated as a linear program [22, eq. 4.5-4.8]. Let V be an $N \times M$ game matrix, and x be the probability distribution over the row player's strategies. The Nash equilibrium distribution for the row player is the solution of the linear program 5.1, where z is the expected score for the equilibrium solution.

As we saw in the game in table 5.1, it is necessary to take the reasoning of

$$
\begin{array}{ll}
\text{Maximize} & z \\
\text{subject to} & \sum_{i=1}^{N} x_i = 1 \\
& z - \sum_{i=1}^{N} V_{i,j} x_i \leq 0 \quad \text{for } j = 1...M \\
& x \geq 0
\end{array}
$$

Figure 5.1: Nash Equilibrium Linear Program

the opponent into account in a zero-sum game. So, to extend the analysis of policy switching to multi-agent systems, we should analyze it in the Markov Game framework. In the next section we review a Markov Game policy switching result and show that it does not provide a strong performance guarantee comparable to the MDP case.

5.2 Switching Theorem

Chang [8] defines a policy switching policy for a minimizing agent in a Markov Game, and shows a bound on how badly the policy switcher can do compared to following the minimax strategy, however this bound can be large. We give an example below in which the policy switcher gets the worst value in the game, while meeting the given bound.

The following is a description of the policy switcher and the error bound. Let Π and Φ be finite sets of stationary policies of a minimizing and a maximizing player in a 2-person Markov game with states $s \in S$. *Minimax policy switching*

policy π_{ps} is defined as

$$\pi_{ps}(s) \in \Big\{ \arg\min_{\pi \in \Pi} \Big(\max_{\phi \in \Phi} V(\pi, \phi)(s) \Big)(s) \Big\}, s \in S. \tag{5.2}$$

$\pi_{ps}(s)$ is a policy that achieves $\min_{\pi \in \Pi} \max_{\phi \in \Phi} V(\pi, \phi)(s)$. Chang shows that

$$\max_{\phi \in \Phi} V(\pi_{ps}, \phi)(s) \leq \min_{\pi \in \Pi} \max_{\phi \in \Phi} V(\pi, \phi)(s) + \frac{\gamma \epsilon}{1 - \gamma}, s \in S \tag{5.3}$$

where $\frac{\gamma \epsilon}{1-\gamma}$ is an error bound in terms of the discount γ and the "degree of local equilibrium" ϵ. ϵ is defined as

$$\epsilon = \max_{s \in S} \Big(\min_{\pi \in \Pi} \max_{\phi \in \Phi} V(\pi, \phi)(s) - \min_{\phi \in \Phi} \min_{\pi \in \Pi} V(\pi, \phi)(s) \Big). \tag{5.4}$$

So ϵ is the maximum difference between the lowest cost the minimizer can expect to secure and the maximizer's worst case. ϵ cannot be made arbitrarily small, and in many games it will be quite large. Next we give an example in which ϵ is as large as possible.

Consider a Markov game with three states s_1, s_2, s_3, deterministic transitions $s_1 \to s_2 \to s_3$, and action costs C given in tables 5.3,5.4.

$C(s_1)$	ϕ_0	ϕ_1
π_0	5	-5
π_1	-4	-4

Table 5.3: Costs at State s_1

$C(s_2)$	ϕ_0	ϕ_1
π_0	-5	5
π_1	4	4

Table 5.4: Costs at State s_2

Let player 1 (row player) be the minimizer, and player 2 (column player) be

the maximizer. The game value matrix $V(\pi_i, \phi_j)(s_1)$ is $C(s_1) + C(s_2)$, so the value of state s_1 is zero for all action pairs, as shown in table 5.5.

$V(s_1)$	ϕ_0	ϕ_1	minmax
π_0	0	0	*
π_1	0	0	*
maxmin	*	*	

Table 5.5: Game Value at State s_1

$V(s_2)$	ϕ_0	ϕ_1	minmax
π_0	-5	5	
π_1	4	4	*
maxmin		*	

Table 5.6: Game Value at State s_2

Since the first choice for the minimizer is arbitrary, assume it chooses π_0, and the maximizer chooses ϕ_0. In state s_2, the minimizer switches to π_1, because this gives the minimax value 4, while the maximizer stays committed to ϕ_0. The final game values for the possible choice combinations are given in table 5.7.

	ϕ_0,ϕ_0	ϕ_1,ϕ_1	criteria
π_0,π_0	0	0	
π_0,π_1	**9**	-1	policy switching
π_1,π_0	-9	1	
π_1,π_1	0	0	policy switching

Table 5.7: Cost Sums for Action Sequence Pairs

In this example, the policy switching minimizer received its worst possible result, 9. This happened because the policy switcher made inconsistent assumptions about the opponent. The policy switcher looks forward by assuming that the opponent stays with a policy throughout the game, but in state 2 the policy switcher avoids the threat of the opponent getting a reward of 5 by choosing ϕ_1, even though it could observe that the maximizer chose ϕ_0 in state 1.

For this game, the equilibrium term is

$$\epsilon = \max_{s \in S}(\min_{\pi} \max_{\phi} V(\pi, \phi)(s) - \min_{\phi} \min_{\pi} V(\pi, \phi)(s)) \tag{5.5}$$

$$= \max\{0 - 0, 4 - (-5)\} \tag{5.6}$$

$$= 9 \tag{5.7}$$

So in this example, the equilibrium error term ϵ is as large as possible, and its value is achieved by the switching policy. So in the Markov Game framework, a policy switching agent can underperform the best single policy. In contrast, it has been shown that policy switching in an MDP is guaranteed to do no worse than any single policy.

5.3 Monotone Maximin Strategy Switching

To address the weakness in maximin strategy switching, we define monotone switching. A monotone player switches strategies only when the maximin value plus the reward to that point exceeds the largest maximin and reward previously seen, so it should not be misled into a lower value choice. The minimax version of the monotone selection algorithm is shown in figure 5.2.

Next, we compare the performance of monotone switching to minimax switching using the game given in section 5.2. The monotone value at s_1 is 0 (accumulated cost plus minimax). The monotone values for different choices at state s_2 are given in table 5.8. The monotone player will choose the minimax policy at s_2 only if

v' is smallest previous minimax plus accumulated cost
$c \leftarrow$ accumulated cost
$v \leftarrow \min_{\pi_i} \max_{\phi_j} V(\pi_i, \phi_j)$
if $v + c < v'$
  then $v' \leftarrow v + c$
    select $\arg\min_{\pi_i} \max_{\phi_j} V(\pi_i, \phi_j)$
  else select previous strategy

Figure 5.2: Monotone Minimax Selection

	ϕ_0	ϕ_1
π_0	$5+4=9$	$-5+4=-1$
π_1	$-4+4=0$	$-4+4=\ \ 0$

Table 5.8: Monotone Values at s_2 Based on Choices at s_1

the monotone value is less than 0. The possible action sequences and the maximin and monotone player choices are shown in table 5.9

	ϕ_0,ϕ_0	ϕ_1,ϕ_1	criteria
π_0,π_0	$0^\dagger$	0	†monotone
π_0,π_1	9	$-1^\ddagger$	policy switching, ‡monotone
π_1,π_0	-9	1	
π_1,π_1	0	0	policy switching, monotone

Table 5.9: Cost Sums for Action Sequence Pairs

Monotone and minimax players choose the same strategies when they start by choosing π_1. If the first choice was π_0, the minimax player switches to π_1 and can receive a cost of 9. But the monotone player only switches when the opponent starts by choosing ϕ_1. The monotone player outperforms the minimax player and meets or exceeds the minimax value calculated at state s_1.

5.4 Strategy Switching Planners

The only requirement of an implementation of the planner interface is that given a state it returns a strategic plan. Our basic planner, the `GoalDrivenPlanner`, uses a single strategy to generate a plan. A planner might be able to improve on the performance of any single strategy by switching among strategies in a set. We developed three switching planners using maximin, Nash, and monotone strategy selection to test planning by strategy switching. The first thing the switching planner must do is build a game matrix of the estimated score for all pairs of player-opponent strategies. We assume the player and the opponent both have the same strategy choices, and we generate the game matrix as follows: for each strategy pair, generate a plan from each strategy, simulate the plan for player and opponent for a planning cycle, replan using the same strategies, and continue simulation and replanning epochs to the end of the game. The final score from the simulation becomes the game matrix entry for the strategy pair. After the matrix has been completed, the planner selects a strategy. The switching planner returns the plan generated from the selected strategy to the controller.

Assuming a strategy set of 10 strategies, there are 100 games to simulate to calculate a game matrix. The simulator can complete 100 games in about 2 seconds. The matrix simulation could be done concurrently with the game play, allowing near real-time decisions, though currently the client is single-threaded, so there is a short interruption to the game at each planning epoch.

The architecture of the switching planner is shown in figure 5.3. The switching

planner uses a controller implemented by the `SimController` class that plays the same role in the planner as the `StrategyController` does in the client. The `SimController` has a pair of `StrategyManagers` that manage the execution of generated player and opponent plans in the simulated game. An important feature of our architecture is that plans can be executed in the simulator or the engine with only small modifications to the `StrategyManager`. Since the plan generator works with the abstract state as input, the same code is used to generate plans for the engine state as for the simulated state.

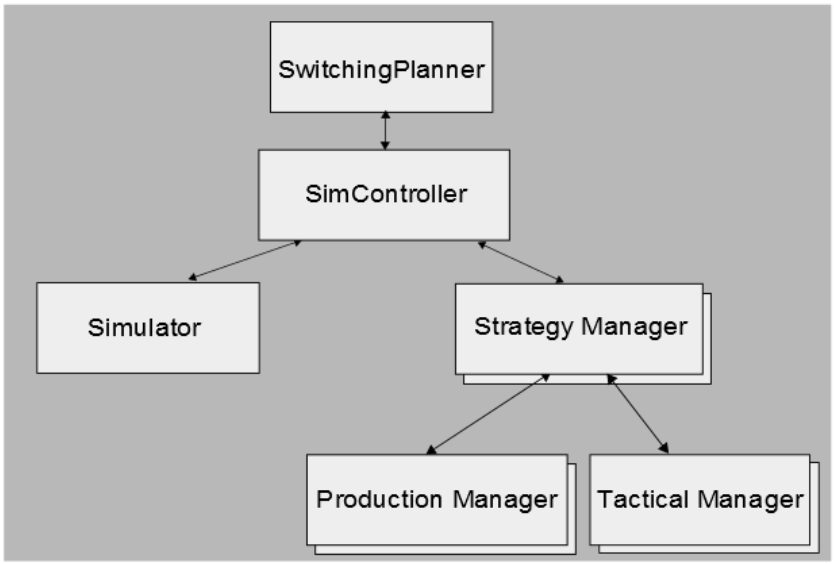

Figure 5.3: Switching Planner and Simulator

Chapter 6 – Results

6.1 Experimental Setup

6.1.1 Strategy Set

Switching planners work by simulating games between players who use strategies from a given strategy set. The simulated games produce a matrix of player vs. opponent game values which can be treated as a strategic form game. Using a criterion such as Nash equilibrium or maximin, the switching planner selects a strategy, generates a plan from the strategy, and returns the plan to the controller. The strategy set used for the switching players is named "2012-02-05", and its parameters are given in table 6.1 (The "rush" strategies given here are not really rushes, since they use large groups. It might be better to call them"aggressive" strategies).

A goal is a type of region to attack or secure. The goal types are "base", "enemy", and "chokepoint". The plan generator creates tasks so that the higher priority goals are pursued first. A priority zero goal is ignored. The "Units Per Goal" parameter specifies the size of the group that the controller should produce and send to the goal region.

As part of the experiment, switching planners compete against the Wargus built-in AI script. Since complicated builds were not part of the strategies available

	Goal Priority			Units Per Goal		
	base	enemy	chkpt.	base	enemy	chkpt.
0. balanced 7	1	3	2	7	7	7
1. balanced 7 mass	1	3	2	7	7	7
2. balanced 9	1	3	2	9	9	9
3. balanced 9 mass	1	3	2	9	9	9
4. rush 7	1	3	2	5	7	3
5. rush 7 mass	1	3	2	5	7	3
6. rush 9	1	3	2	7	9	4
7. offensive 5 mass	0	1	0	0	5	0
8. offensive 7 mass	0	1	0	0	7	0
9. chokepoint 6	2	1	3	6	6	6

Table 6.1: Strategy Set 2012-02-05 Definition.

to planners, the scripts for the built-in Wargus AI player were adjusted by removing the unit upgrades.

6.1.2 Data Sets

To evaluate the performance of the strategy switching planners, we gathered statistics on strategy pairs playing against each other in simulation and in the Stratagus engine. In simulated games, the plans returned by planners are executed in the simulator. Strategy switching planners use an inner simulation to predict the results of games played by strategy pairs. In these games, switching planners make perfect predictions. For the second data set, plans are executed by sending actions to the Stratagus engine. Switching planners still use simulation to make predictions, but in this case the predictions are imperfect.

Games were played on two maps that were prepared to present different chal-

lenges to the planners. Maps had two starting positions, and the planners played games from both positions on the map, so for each pair there were four map configurations to play. Because of randomness in Wargus game play, each combination of players and map were run multiple times to gather performance statistics. These combinations are summarized in tables 6.2 and 6.3. Table 6.2 shows there were 10,000 episodes (games) of fixed strategy versus fixed strategy games played. For 10 fixed strategies there are 50 pairs, disregarding order. Each pair played 50 episodes in 4 configurations, giving a total of 10,000 episodes. In the switching vs. switching games there was no self-play, so there were 3 pairs. These were played for 30 episodes on each of 4 maps giving 360 episodes. For strategy pairs played in the Wargus engine, we ran each pair until one player won, one player achieved 3 times the hitpoints of the opponent, or until 80,000 cycles were completed.

player	opponent	pairs	episodes	configs	episodes
fixed	fixed	50	50	4	10,000
switching	fixed	30	50	4	6,000
switching	other switching	3	30	4	360
switching	built-in	3	30	4	360
				Total	16,720

Table 6.2: Stratagus Data Sets

Since our simulations are deterministic, only one simulated game was played for each pair and map combination. In simulation, we run the full 100 pairs of fixed strategies, because they can be run in a few seconds.

player	opponent	pairs	configs	episodes
fixed	fixed	100	4	400
switching	fixed	30	4	120
		Total	520	

Table 6.3: Simulation Data Sets

6.1.3 Map Design

We chose two maps from the set packaged with Wargus to present different problems to the planners. The `2bases` map is the `one-way-in-one-way-out` map packaged with Wargus and initialized with two production bases for each player. This meant that there was a difference between a mass attack and a dispersed attack on this map. The other map was `the-right-strategy` map initialized with one base for each player. On this map there is no difference between mass attack and dispersed attack strategies, but it has narrow passages between the opposing bases, so this map was more of a challenge for tactical unit control. The Wargus "minimap" view of `2bases` and the strategic abstraction used by the planners are shown in figures 6.1 and 6.2. The minimap for `the-right-strategy` and the corresponding strategic map are shown in figures 6.3 and 6.4.

Each player starts with enough buildings to create combat groups capable of destroying the opponent. We made the maps fair by adjusting the positions of buildings until the maximin vs. maximin planner pair achieved a near 50% win rate from both positions on the map.

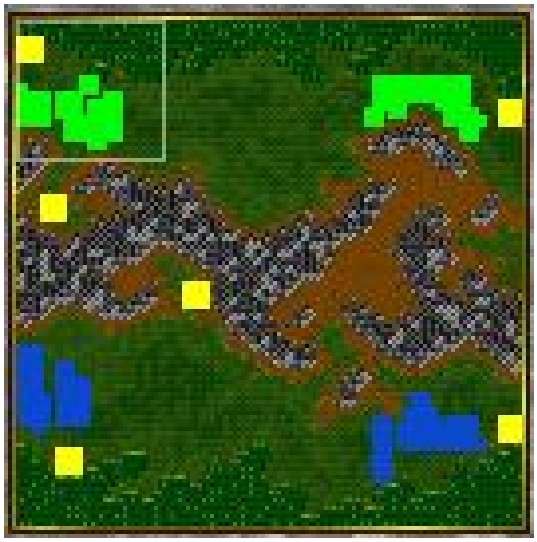

Figure 6.1: `2bases` Minimap

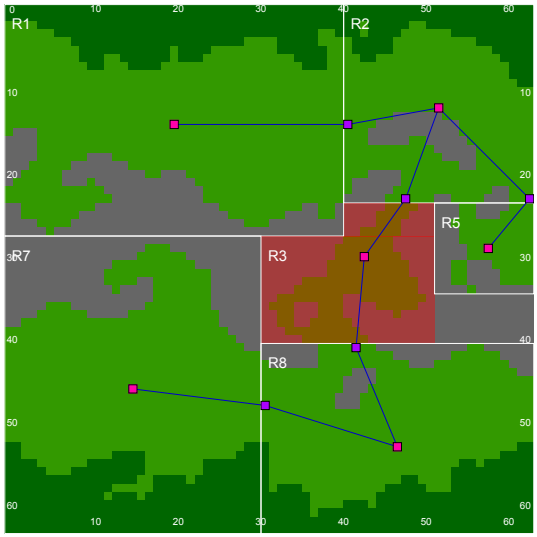

Figure 6.2: `2bases` Strategic Map

Figure 6.3: `the-right-strategy` Minimap

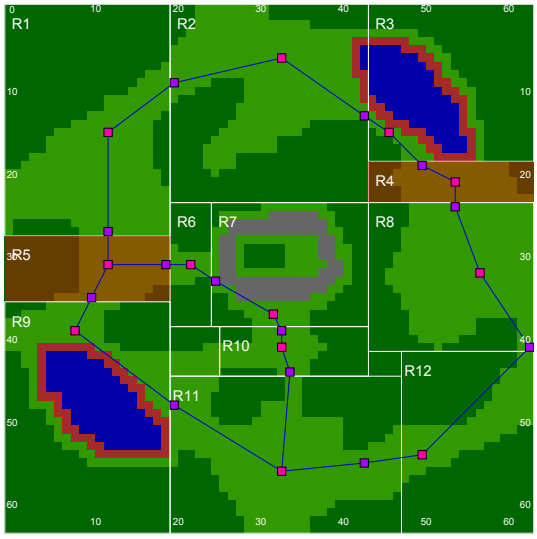

Figure 6.4: `the-right-strategy` Strategic Map

6.2 Results

6.2.1 Maximin and Minimums

To evaluate the performance of the switching planners, we compare the minimum scores achieved by switching planners to the maximin of the fixed strategy set. To be worthwhile, a switching planner should do no worse than the fixed strategy maximin value, though we saw in section 5.2 that it is possible for a switching agent to get a worse value than a fixed strategy agent. Scores and win rates in tables in this section are given from the column player's perspective, so we can use space at the bottom of the table for summarizing results. The maximin in simulation is shown in tables 6.4 and 6.5. The maximin of the mean scores in the engine are shown in tables 6.6 and 6.7. In these sets of games the maximin values are found on the diagonals where the large group strategies are played against themselves. The maximin strategies are 3 (`balanced 9 mass`) in simulation and in the engine, and 2 (`balanced 9`) and 6 (`rush 9`) in simulation. These maximin strategies are the best risk-averse strategies in the initial state. In simulation these values are almost all zero, but scores vary in the engine due to randomness in game play. The win rates for strategies in the strategy set are shown in tables 6.8 and 6.9. The diagonal values are near 50% as expected.

To find the switching planner minimums, we calculate their scores against each fixed strategy on each map. The scores in simulation are shown in tables 6.10 and 6.11. The mean scores in the engine are shown in tables 6.12 and 6.13. The switching planners have similar performance in simulation although the `monotone`

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
0.	**0**	0	8282	8200	-8251	-8251	8262	-8238	-8178	-7238
1.	0	**0**	8282	8200	-8289	-8289	8262	-8204	-8172	-7238
2.	-8186	-8186	**0**	0	-6247	-6247	7541	-7546	-8338	-8335
3.	-8200	-8200	0	**0**	-6199	-6199	7541	-7538	-8248	-8231
4.	5659	5460	8265	8494	**3806**	3806	8205	-8134	-8265	-4625
5.	5659	5460	8265	8494	3806	**3806**	8205	-8104	-8225	-4625
6.	-8130	-8130	-6718	-6718	-8550	-8550	**0**	-7599	-8378	-7864
7.	8350	8288	7846	7846	8246	8188	7802	**0**	7812	7746
8.	8178	8182	8346	8256	8280	8159	8366	-7848	**0**	5189
9.	8281	8299	8320	8258	8137	8139	8292	-8478	7727	**0**
min.	-8200	-8200	-6718	-6718	-8550	-8550	0	-8478	-8378	-8335
maxmin							*			

Table 6.4: Strategy Simulation Scores on `2bases`

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
0.	**0**	0	5542	5542	-5294	-5294	5602	-5608	-6083	-5548
1.	0	**0**	5542	5542	-5294	-5294	5602	-5608	-6083	-5548
2.	-5548	-5548	**0**	0	-5707	-5707	-5579	-5453	-5628	-5596
3.	-5548	-5548	0	**0**	-5707	-5707	-5579	-5453	-5628	-5596
4.	5160	5160	5410	5410	**0**	0	5467	-5480	-5712	-5371
5.	5160	5160	5410	5410	0	**0**	5467	-5480	-5712	-5371
6.	-5548	-5548	5737	5737	-5613	-5613	**0**	-5493	-5668	-5548
7.	5604	5604	5559	5559	5480	5480	5531	**0**	5296	5808
8.	6107	6107	5568	5568	5700	5700	5668	-5296	**0**	-5294
9.	-5691	-5691	5541	5541	-5754	-5754	5581	-5764	-267	**0**
min.	-5691	-5691	0	0	-5754	-5754	-5579	-5764	-6083	-5596
maxmin			*	*						

Table 6.5: Strategy Simulation Scores on `the-right-strategy`

planner gets the highest score. The minimums on `2bases` occur when the switching planners play against the `balanced 9` and `balanced 9 mass` strategies. The highest score is achieved by the `Nash` planner. On `the-right-strategy` map the `monotone` planner does much better than the other two. Notably, the `monotone`

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
0.	**18**	267	5358	3730	-6687	-5451	3525	-8528	-7906	-7672
1.	-267	**2000**	5010	4466	-6014	-5814	2909	-8457	-7925	-7773
2.	-5358	-5010	**127**	780	-8238	-8014	-1023	-8647	-8403	-8443
3.	-3730	-4466	-780	**-630**	-7646	-7738	-2142	-8633	-8484	-8335
4.	6687	6014	8238	7646	**-142**	-546	7585	-8347	-6868	-271
5.	5451	5814	8014	7738	546	**-929**	7026	-8220	-7279	-3111
6.	-3525	-2909	1023	2142	-7585	-7026	**-1001**	-8460	-8440	-7637
7.	8528	8457	8647	8633	8347	8220	8460	**-1526**	7782	8115
8.	7906	7925	8403	8484	6868	7279	8440	-7782	**356**	1352
9.	7672	7773	8443	8335	271	3111	7637	-8115	-1352	**638**
min.	-5358	-5010	-780	-630	-8238	-8014	-2142	-8647	-8484	-8443
maxmin				*						

Table 6.6: Strategy Mean Scores on `2bases`

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
0.	**-44**	545	4328	3387	-3404	-3862	2476	-5736	-5158	-4614
1.	-545	**-1250**	4498	4123	-3235	-4009	1101	-5814	-5256	-5116
2.	-4328	-4498	**-502**	663	-5701	-5834	-1524	-5987	-5878	-5755
3.	-3387	-4123	-663	**-86**	-5883	-5376	-1667	-6000	-5867	-5953
4.	3404	3235	5701	5883	**866**	311	5525	-5664	-4913	1190
5.	3862	4009	5834	5376	-311	**254**	4824	-5681	-4847	1970
6.	-2476	-1101	1524	1667	-5525	-4824	**-360**	-5866	-5629	-4219
7.	5736	5814	5987	6000	5664	5681	5866	**-996**	4833	4408
8.	5158	5256	5878	5867	4913	4847	5629	-4833	**2**	2795
9.	4614	5116	5755	5953	-1190	-1970	4219	-4408	-2795	**-410**
min.	-4328	-4498	-663	-86	-5883	-5834	-1667	-6000	-5878	-5953
maxmin				*						

Table 6.7: Strategy Mean Scores on `the-right-strategy`

planner gets its highest score where the other two get their minimums, against `balanced 9`.

The opponents that do best in simulation, `balanced 9` and `balanced 9 mass`, also do best in games in the engine. However the `monotone` planner does not do as

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
0.	**50%**	52%	81%	73%	11%	17%	71%	0	2%	5%
1.	48%	**62%**	79%	77%	12%	15%	67%	0	2%	3%
2.	19%	21%	**52%**	56%	0	2%	43%	0	1%	1%
3.	27%	23%	44%	**46%**	3%	2%	35%	0	0	1%
4.	89%	88%	100%	97%	**50%**	48%	97%	0	8%	51%
5.	83%	85%	98%	98%	52%	**44%**	92%	0	4%	31%
6.	29%	33%	57%	65%	3%	8%	**46%**	0	0	4%
7.	100%	100%	100%	100%	100%	100%	100%	**40%**	100%	100%
8.	98%	98%	99%	100%	92%	96%	100%	0	**52%**	61%
9.	95%	97%	99%	99%	49%	69%	96%	0	39%	**56%**

Table 6.8: Strategy Win Rate on `2bases`

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.
0.	**50%**	54%	88%	78%	19%	16%	72%	1%	6%	10%
1.	46%	**40%**	91%	85%	21%	13%	58%	0	6%	5%
2.	12%	9%	**46%**	56%	2%	1%	36%	0	1%	1%
3.	22%	15%	44%	**50%**	0	5%	37%	0	1%	0
4.	81%	79%	98%	100%	**58%**	53%	98%	0	3%	60%
5.	84%	87%	99%	95%	47%	**52%**	92%	0	6%	67%
6.	28%	42%	64%	63%	2%	8%	**50%**	0	2%	13%
7.	99%	100%	100%	100%	100%	100%	100%	**40%**	96%	88%
8.	94%	94%	99%	99%	97%	94%	98%	4%	**50%**	74%
9.	90%	95%	99%	100%	40%	33%	87%	12%	26%	**48%**

Table 6.9: Strategy Win Rate on `the-right-strategy`

well as predicted on `the-right-strategy` map, and in fact performs worse than the other two switching planners.

	Nash	maximin	monotone
0. balanced 7	8128	8128	8128
1. balanced 7 mass	8173	8128	8128
2. balanced 9	8122	8122	8122
3. balanced 9 mass	8122	8122	8122
4. rush 7	8486	8562	8562
5. rush 7 mass	8522	8562	8562
6. rush 9	8570	8256	8404
7. offensive 5 mass	8439	8439	8439
8. offensive 7 mass	8353	8353	8353
9. chokepoint 6	8260	8256	8256
mean	8317	8293	8307
minimum	8122	8122	8122

Table 6.10: Switching Planner Scores in Simulation on 2bases

	Nash	maximin	monotone
0. balanced 7	5545	5525	5547
1. balanced 7 mass	5545	5547	5535
2. balanced 9	1770	2702	6066
3. balanced 9 mass	3818	4018	6066
4. rush 7	5634	5648	5594
5. rush 7 mass	5558	5594	5648
6. rush 9	5658	5375	5375
7. offensive 5 mass	5506	5506	5506
8. offensive 7 mass	5598	5598	5598
9. chokepoint 6	5568	5568	5568
mean	5020	5108	5650
minimum	1770	2702	5375

Table 6.11: Switching Planner Scores in Simulation on the-right-strategy

	Mean Scores			Win Rates		
	Nash	maximin	monotone	Nash	maximin	monotone
0. balanced 7	3767	4064	3120	72%	74%	67%
1. balanced 7 mass	2506	3750	3095	64%	72%	68%
2. balanced 9	-1018	-73	38	42%	50%	52%
3. balanced 9 mass	-1436	-242	-749	41%	49%	45%
4. rush 7	7151	7262	7531	93%	94%	96%
5. rush 7 mass	7318	6932	7630	95%	91%	96%
6. rush 9	560	774	643	56%	54%	54%
7. offensive 5 mass	8508	8559	8476	100%	100%	100%
8. offensive 7 mass	8464	8325	8521	100%	99%	100%
9. chokepoint 6	7988	8031	7765	97%	97%	96%
mean	4381	4738	4607			
minimum	-1436	-242	-749			

Table 6.12: Switching Planner Mean Scores on 2bases

	Mean Scores			Win Rates		
	Nash	maximin	monotone	Nash	maximin	monotone
0. balanced 7	3451	2956	3395	79%	74%	79%
1. balanced 7 mass	3608	3945	4518	81%	85%	90%
2. balanced 9	-277	-588	-814	48%	45%	42%
3. balanced 9 mass	-303	-387	-724	48%	46%	43%
4. rush 7	5649	5338	5505	98%	96%	97%
5. rush 7 mass	5035	4944	5621	93%	92%	99%
6. rush 9	437	1592	451	54%	65%	54%
7. offensive 5 mass	6005	5942	5975	100%	100%	100%
8. offensive 7 mass	5783	5668	5509	98%	97%	96%
9. chokepoint 6	5431	5747	5371	96%	99%	95%
mean	3482	3516	3481			
minimum	-303	-588	-814			

Table 6.13: Switching Planner Mean Scores on the-right-strategy

Next, we compare the fixed strategy maximin to the switching planner minimums using results from the previous tables. The results in simulation are shown in table 6.14. The results in the engine are shown in table 6.15.

	Fixed	`Nash`	`maximin`	`monotone`
`2bases`	0	8122	8122	8122
`the-right-strategy`	0	1770	2702	5375

Table 6.14: Fixed Strategy Maximin and Switching Planner Minimums in Simulation

	Fixed	`Nash`	`maximin`	`monotone`
`2bases`	-630	-1436	-242	-749
`the-right-strategy`	-86	-303	-588	-814

Table 6.15: Switching Planner Minimum Means in Engine

The minimum values achieved by the switching planners in simulation are above the fixed strategy maximin. In the engine, the `maximin` switching planner performs better than the maximin value on the `2bases` map, but no switching planner achieves the maximin value on `the-right-strategy`.

Since there is some randomness in the effects of actions in Wargus, we should consider confidence intervals for the results. In tables 6.16 and 6.17, we show the 95% confidence interval bounds given by the Wilson score interval around the win rate. Here we take each game as a Bernoulli trial where the probability of success is the win rate. The Wilson score confidence interval is widest at 50%, and with 100 samples we get interval bounds of about $\pm 10\%$. Since the maximin value occurs for a strategy vs. self game, the win rate is near 50% and the confidence interval

is wide. The confidence intervals around the minimum means contain the fixed strategy maximin mean.

	Player	Opponent	Score	Rate Confidence
maximin	balanced 9 mass	balanced 9 mass	-630	46% (33%,60%)
minimums	Nash	balanced 9 mass	-1436	41% (32%,51%)
	maximin	balanced 9 mass	-242	49% (39%,59%)
	monotone	balanced 9 mass	-749	45% (36%,55%)

Table 6.16: Strategy Pairs on `2bases`

	Player	Opponent	Score	Rate Confidence
maximin	balanced 9 mass	balanced 9 mass	-86	50% (37%,63%)
minimums	Nash	balanced 9 mass	-303	48% (38%,58%)
	maximin	balanced 9	-588	45% (36%,55%)
	monotone	balanced 9	-814	42% (33%,52%)

Table 6.17: Strategy Pairs on `the-right-strategy`

6.2.2 Simulation Accuracy

Simulations must be accurate to be useful for strategy switching. Figures 6.5 and 6.6 compare scores achieved by strategies in simulation to the mean scores achieved in the engine. The strategy simulations are accurate enough to predict wins or losses and to closely predict average game scores, except strategies 0 (`balanced 7`) and 1 (`balanced 7 mass`) on `the-right-strategy` map. The simulator overestimates the strength of the wins for the strategy 6 (`rush 9`) on `2bases`, and 2 (`balanced 9`) and 3 (`balanced 9 mass`) on `the-right-strategy`. These are the strategies with the largest groups.

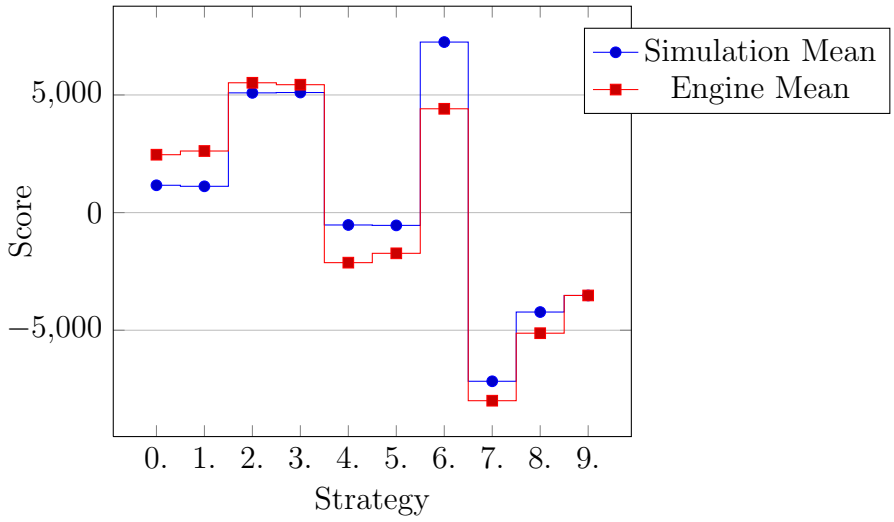

Figure 6.5: Scores in Simulation and Engine on `2bases`

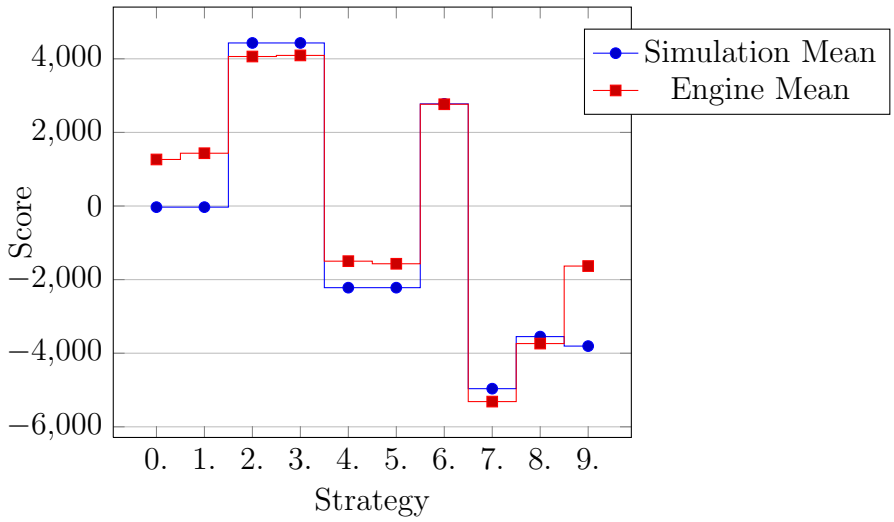

Figure 6.6: Scores in Simulation and Engine on `the-right-strategy`

6.2.3 Switching Planner Choices

To help assess how effective the switching algorithms are, we should examine the strategies being selected over time. Are the switching planners winning simply by favoring a few strong strategies, and are the other strategies irrelevant? Table 6.18 show that about 65-75% of the switching planner choices are made from 3 strategies that favor large groups, although the mixes are different, and the `Nash` planner uses the `balanced 7` strategy much more that the other two. But table 6.19 shows that the mix of strategies chosen by the `maximin` planner changes as games progress. In games that last longer than 20,000 cycles, the planner begins to favor smaller groups. For example the rate of choosing `offensive 5` increases from 0% to 6% by cycle 80,000. The planner also begins to prioritize controlling chokepoints in longer games. This shows that the planner is responding to differences in the game state and that the strategy set does have a useful mix of strategies.

	`Nash`	`maximin`	`monotone`
0. `balanced 7`	22%	6%	7%
1. `balanced 7 mass`	5%	5%	3%
2. `balanced 9`	34%	26%	30%
3. `balanced 9 mass`	3%	16%	17%
4. `rush 7`	3%	4%	2%
5. `rush 7 mass`	1%	4%	3%
6. `rush 9`	19%	25%	26%
7. `offensive 5 mass`	5%	4%	2%
8. `offensive 7 mass`	2%	4%	3%
9. `chokepoint 6`	7%	7%	6%

Table 6.18: Strategy Choices of Switching Planners

Another factor that may contribute to the lower scores of the switching planners

	6,030	12,060	18,090	24,120	80,000
0. `balanced 7`	0	8%	7%	7%	8%
1. `balanced 7 mass`	0	6%	6%	6%	9%
2. `balanced 9`	23%	33%	29%	28%	18%
3. `balanced 9 mass`	27%	15%	13%	12%	7%
4. `rush 7`	0	5%	5%	5%	5%
5. `rush 7 mass`	0	5%	4%	6%	6%
6. `rush 9`	50%	16%	12%	13%	17%
7. `offensive 5 mass`	0	3%	6%	5%	6%
8. `offensive 7 mass`	0	3%	8%	8%	6%
9. `chokepoint 6`	0	6%	9%	10%	17%

Table 6.19: `maximin` Choices by Epoch

on the `the-right-strategy` map are shown in tables 6.20 and 6.21. Table 6.20 shows that the `maximin` planner uses large groups in the early game (`balanced 9`, `balanced 9 mass`, and `rush 9`), then quickly switches to prioritizing smaller offensive groups and holding the chokepoints. This switching appears effective, because no game lasts to 80,000 cycles. In contrast, in the engine the `maximin` planner consistently favors the large group strategies, and games last to the maximum limit of 80,000 cycles. By watching games in the engine, we have seen that the path finding algorithms do poorly at managing large groups of units. The units get stuck in cul-de-sacs, block each other in narrow passages, and get dispersed when traveling long distances. This is particularly true on the `the-right-strategy` map, because of the long, narrow passages by water. The simulator overestimates the effectiveness of the large group strategies, and the switching planners persist in using them. Figure 6.6 confirms this observation, because it shows that strategies 2 (`balanced 9`) and 3 (`balanced 9 mass`) are overestimated.

	6,030	12,060	18,090	24,120	80,000
0. `balanced 7`	0	0	0	0	0
1. `balanced 7 mass`	0	0	0	100%	0
2. `balanced 9`	50%	0	0	0	0
3. `balanced 9 mass`	50%	0	50%	0	0
4. `rush 7`	0	0	0	0	0
5. `rush 7 mass`	0	0	0	0	0
6. `rush 9`	0	0	0	0	0
7. `offensive 5 mass`	0	0	50%	0	0
8. `offensive 7 mass`	0	50%	0	0	0
9. `chokepoint 6`	0	50%	0	0	0

Table 6.20: `maximin` vs. `balanced 9` Choices in Simulator

	6,030	12,060	18,090	24,120	80,000
0. `balanced 7`	0	5%	9%	10%	12%
1. `balanced 7 mass`	0	2%	4%	5%	9%
2. `balanced 9`	47%	27%	15%	17%	12%
3. `balanced 9 mass`	53%	24%	24%	24%	15%
4. `rush 7`	0	4%	6%	7%	0
5. `rush 7 mass`	0	6%	3%	7%	12%
6. `rush 9`	0	16%	19%	19%	24%
7. `offensive 5 mass`	0	4%	9%	2%	9%
8. `offensive 7 mass`	0	5%	3%	7%	0
9. `chokepoint 6`	0	7%	7%	2%	9%

Table 6.21: `maximin` vs. `balanced 9` Choices

In section 6.2.1 we saw that the `monotone` planner outperformed the fixed strategy planners and the other switching planners in simulation on the `the-right-strategy` map, but in the engine it did not reach the maximum value and performed worse than the other switchers. In tables 6.22 and 6.23 we see the `monotone` planner has the same problem as the `maximin` planner. The `balanced 9` strategies appear more effective in simulation than they are in the engine, so the `monotone` planner

relies on them in the game, but they do not perform as expected. Because of the `monotone` criteria, the `monotone` planner chooses the `balanced 9` strategies even more often than the `maximin` planner does.

	6,030	12,060	18,090	24,120	80,000
0. `balanced 7`	0	0	0	0	0
1. `balanced 7 mass`	0	0	0	0	0
2. `balanced 9`	50%	0	0	0	0
3. `balanced 9 mass`	50%	50%	0	0	0
4. `rush 7`	0	0	0	0	0
5. `rush 7 mass`	0	0	0	0	50%
6. `rush 9`	0	0	50%	100%	0
7. `offensive 5 mass`	0	0	50%	0	0
8. `offensive 7 mass`	0	0	0	0	0
9. `chokepoint 6`	0	50%	0	0	50%

Table 6.22: `monotone` vs. `balanced 9` Choices on `the-right-strategy` in Simulation

	6,030	12,060	18,090	24,120	80,000
0. `balanced 7`	0	5%	5%	2%	16%
1. `balanced 7 mass`	0	4%	3%	0	0
2. `balanced 9`	59%	28%	32%	33%	23%
3. `balanced 9 mass`	41%	28%	29%	36%	49%
4. `rush 7`	0	3%	2%	0	0
5. `rush 7 mass`	0	2%	5%	7%	2%
6. `rush 9`	0	12%	8%	7%	2%
7. `offensive 5 mass`	0	7%	5%	2%	5%
8. `offensive 7 mass`	0	1%	2%	2%	0
9. `chokepoint 6`	0	10%	12%	10%	3%

Table 6.23: `monotone` vs. `balanced 9` Choices

6.2.4 Switching Planner Performance

Finally, we want to compare the performance of the three switching planners to each other and to the Wargus built-in AI script. Tables 6.24 and 6.25 show the win rates for switching planners on `2bases` and `the-right-strategy` maps.

	Nash	maximin	monotone
Nash		43%	50%
maximin	57%		48%
monotone	50%	52%	
built-in	100%	100%	100%

Table 6.24: Switching vs. Switching Win Rates on `2bases`

	Nash	maximin	monotone
Nash		42%	53%
maximin	58%		42%
monotone	47%	58%	
built-in	58%	57%	57%

Table 6.25: Switching vs. Switching Win Rates on `the-right-strategy`

The switching planners win all games against the built-in AI script on `2bases`, and 57-58% of games on `the-right-strategy`. The `Nash` planner does well against the `maximin` planner on both maps. The `monotone` planner is a nearly even match for the other two on `2bases`, but loses to the `maximin` player on `the-right-strategy`, which contrasts with how well it does in simulation.

Chapter 7 – Summary and Future Work

> If the art of war were nothing but the art of avoiding risks, glory would become the prey of mediocre minds.
>
> Napoléon Bonaparte

Our work has shown that a strategy switching planner can play a complete RTS game that requires coordinating production and combat. The strategy switching planners performed well against the Stratagus built-in AI, and they came close to reaching the maximin value of the planner's set of fixed strategies. There are two main contributions from this work. The first contribution is the grammars for game abstraction and strategic plans. There are several advantages to having a grammar. The grammars define common strategic concepts for planning and simulation. This allows us to have unified messaging throughout the levels of the manager hierarchy, and the different modules share a common strategic view. Using the grammar, the simulator and game controllers speak the same language, so few changes are needed for using the top-level controller code in both contexts. Having state and plan grammars also means we can save state and plan descriptions in text files at decision points in the game, which greatly helps in analyzing planner performance.

The second contribution is to show the theoretical weaknesses of adversarial planning by strategy switching. Strategy switching in a multi-player Markov Game does not guarantee better performance than following the best single strategy in a set. Although the strategy switching planners performed better than fixed strategy planners in simulation, they did not reach the maximin value in the engine games, and we have shown that strategy switching controllers can be misled into sub-optimal choices.

Several different architectures have been used for incorporating AI players into RTS games with some form of scripting being the most popular. In our system there is no triggering of scripts from recognized states. Rather, an abstraction of the current state leads to goal formation, plans are generated from a strategy template, and tasks are distributed to a hierarchy of unit group managers. This frees the developer to write code at a higher level of abstraction, and since the controller does not rely on fixed action sequences, we believe it will perform well in a wider range of game scenarios.

In future work we would like to extend the game system to cover all aspects of the game. The division of the maps into regions was coded by hand, and it was a process of trial and error to represent the symmetry of the maps. It would be better if the process of map abstraction (division into regions, creating the connectivity graph, and identifying chokepoints) were automated. Typically an RTS game starts with a single worker per player, but our planners could not start from this state because we have an incomplete implementation of the production manager. We did some early experiments using the resource planner described in [7], but the

main unresolved problem is building placement. As we saw in experiments, simply scanning for empty spaces near a base is inadequate. Naive building placement can result in long paths for workers and combat units, and worse, buildings can block worker and combat unit paths. What is needed for building placement is a tactical level of path analysis, so that buildings do not restrict combat groups, so that buildings are placed as shields against possible enemy attacks, and so that building placement is optimized to balance path lengths to resources and initial placement of combat units. The need to balance various factors suggests an optimization approach such as what was used in the linear programming combat controller. The last capability we need for playing all aspects of the game is to be able to plan with hidden state, known as "fog of war" in RTS games. In Game Theory, games with hidden state are called games of *imperfect information*. We may be able to extend our strategy switching planners to games of imperfect information with some inspiration from work on poker-playing programs [4].

The switching planners relied on an engineered set of strategies. During this project we made two attempts at developing an algorithm for automatically generating strategy sets, but the generated sets all performed worse than the engineered set. An algorithm for finding better sets would be a powerful way to improve the switching planners.

The Nash, maximin, and monotone selection criteria are all risk-averse approaches appropriate when assuming full observability, and an opponent who has knowledge of the full strategy set and perfect rationality. In most games, state is hidden through "fog of war ", the opponent has no knowledge of a player's strategy

set, and the opponent does not have perfect rationality, so it may be better to use less risk-averse selection criteria. Choosing by minimax regret [15], or incorporating a model of risk may make the controller less risk-averse, so that it can take advantage of opportunity.

Our planner was misled by the simulator's overestimates of scores for large group strategies. It might be useful to characterize the effects of overestimates and underestimates. Our planner plans at fixed intervals, so could miss opportunities. It would be better if replanning were triggered by recognition of unexpected state, and recognition of unexpected states could be used to adjust scores provided by the simulator. We would like to overcome the lower performance found on the map that presented tactical movement problems (`the-right-strategy` map). This may come from customizing the tactical path finding algorithm used by units, or by adjustments to the simulator to penalize large group movements. There may be a need for peer-level communication between group managers. Combat groups can block each other's paths, and there has to be a way to resolve such tactical-level conflicts. Rather than putting the burden of detecting group conflicts on the high-level manager, it may be better to allow groups to communicate to resolve such conflicts themselves. Finally, for a better assessment of the performance of the strategy switching planners, we need to test them against human players.

The complete source code is availabe as open source [2].

APPENDICES

Appendix A – Game State Grammar

The ANTLR [19] grammar for the abstract game state is given in figure A.1. The state describes the map, its division into regions, the connectivity graph for the map, and the attributes of the units on the map. Figure A.2 shows a short example state description.

```
game_state : '(' ':GameState' ':cycle' INT player+ map units ')';
player : '(' ':player' ':id' INT pair+ ')';
map  : '(' ':GameMap' cells? region+ cnx? ')'
       | '(' ':GameMap' ':resource' STRING ')';
cells  : '(' ':cells' map_row+ ')';
map_row  : STRING;
region : '(' ':Region' ':id' INT ':center' location chokepoint? rect+  ')';
chokepoint  : ':chokepoint';
location : '(' INT INT ')';
rect : '(' INT INT INT INT ')';
cnx : '(' ':connections' pnode+ passage+ ')';
passage : '(' ':Passage' pair pair ')';
pnode : '(' ':PassageNode' ':id' INT location ')';
units : '(' ':units' unit* ')';
unit : '(' ':Unit' pair+ ')';
pair : LABEL (INT|NAME);
```

Figure A.1: Game State Grammar

```
(:GameState :cycle 50
  (:player :id 0 :gold 400 :oil 500 :wood 600 :supply 0 :demand 0)
  (:GameMap
    (:cells ...(omitted for space)...)
    (:Region :id 1 :center (20 4) (0 0 39 7))
    (:Region :id 2 :center (20 15) (0 10 39 19))
    (:Region :id 3 :center (10 8) (7 8 12 9))
    (:connections
      (:PassageNode :id 1 (10 7))
      (:PassageNode :id 2 (10 10))
      (:Passage :regionNode 1 :passageNode 1)
      (:Passage :regionNode 2 :passageNode 1)
      (:Passage :regionNode 2 :passageNode 2)
      (:Passage :regionNode 3 :passageNode 2)
    )
  )
  (:units
    (:Unit :unitId 1 :ownerId 0 :RAmount 40000 :CurrentTarget -2
             :HitPoints 25500 :LocX 2 :LocY 2 :Status 3 :Armor 20
             :Damage 3 :PiercingDmg 0 :StatusArg1 -2 :StatusArg2 0
             :Type 11 :UnitTypeString unit-gold-mine)
  )
)
```

Figure A.2: Game State Example

Bibliography

[1] Stratagus: A real-time strategy engine. `http://stratagus.sourceforge.net/`.

[2] StratagusAI. `http://beaversource.oregonstate.edu/projects/stratagusai`.

[3] Wargus - warcraft II. `https://launchpad.net/wargus`.

[4] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 661–668, 2003.

[5] M. Buro. Real-time strategy gaines: A new AI research challenge. In *Proceedings of the 18th International Joint Conference on Artificial intelligence*, pages 1534–1535. Citeseer, 2003.

[6] M. Campbell, A.J. Hoane, and F. Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[7] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura. Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling, Providence, Rhode Island*, 2007.

[8] H.S. Chang. On combining multiple heuristic policies in minimax control. In *Proceedings of the 17th International Symposium on Mathematical Theory of Networks and Systems*, 2006.

[9] H.S. Chang, R. Givan, and E.K.P. Chong. Parallel rollout for online solution of partially observable Markov decision processes. *Discrete Event Dynamic Systems*, 14(3):309–341, 2004.

[10] M. Chung, M. Buro, and J. Schaeffer. Monte Carlo planning in RTS games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*. Citeseer, 2005.

[11] J.A. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer Verlag, 1997.

[12] B. Horling and V. Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(04):281–316, 2004.

[13] J.E. Laird. Using a computer game to develop advanced AI. *Computer*, 34(7):70–75, 2001.

[14] F.W. Lanchester. Mathematics in warfare. *The world of mathematics*, 4:2138–2157, 1956.

[15] K. Leyton-Brown and Y. Shoham. Essentials of game theory: A concise multidisciplinary introduction. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2(1):1–88, 2008.

[16] N.J. MacKay. Lanchester combat models. `arXiv:math/0606300`, 2006.

[17] J. McCoy and M. Mateas. An integrated agent for playing real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1313–1318, 2008.

[18] A. Newell, J.C. Shaw, and H.A. Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2(4):320–335, 1958.

[19] T. Parr. ANTLR parser generator. `http://www.antlr.org/`.

[20] F. Sailer, M. Buro, and M. Lanctot. Adversarial planning through strategy simulation. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 80–87. IEEE, 2007.

[21] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[22] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge Univ Pr, 2009.

[23] G. Sierksma. *Linear and integer programming: theory and practice*, volume 1. CRC, 2002.

[24] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.

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

[26] A.M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[27] D. W. Aha, M. Molineaux, and M. Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. *Case-Based Reasoning Research and Development*, pages 5–20, 2005.

[28] B.G. Weber, M. Mateas, and A. Jhala. Applying goal-driven autonomy to StarCraft. In *Proceedings of the Sixth Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[29] B.G. Weber, M. Mateas, and A. Jhala. Case-based goal formulation. In *Proceedings of the AAAI Workshop on Goal-Driven Autonomy*, 2010.

[30] S. Wintermute, J. Xu, and J.E. Laird. SORTS: A human-level approach to real-time strategy AI. *Ann Arbor*, 1001:48109–2121, 2007.