Co-Evolving Real Time Strategy Game Players

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Science and Engineering

by

Christopher E. Miles

Sushil J. Louis, Ph.D. / Dissertation Advisor

August, 2007
THE GRADUATE SCHOOL

We recommend that the dissertation prepared under our supervision by

CHRISTOPHER E. MILES

entitled

Co-Evolving Real Time Strategy Game Players

be accepted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Sushil J. Louis, Ph.D., Advisor

Sergiu Dascalu, Ph.D., Committee Member

Thomas Quint, Ph.D., Committee Member

David Crowther, Ph.D., Graduate School Representative

Marsha H. Read, Ph.D. Associate Dean, Graduate School

August, 2007
ABSTRACT

Co-Evolving Real Time Strategy Game Players

by

Christopher E. Miles

I aim to develop superior real-time strategy (RTS) game players. Most existing Artificially Intelligent (AI) players for RTS games use a variety of hand-coded approaches including expert systems, scripts and decision trees to play the game. These traditional approaches suffer from the knowledge acquisition bottleneck common to many AI systems. To improve on this, I investigate the use of genetic algorithms to co-evolve AI players for real-time strategy games, overcoming the knowledge acquisition bottleneck while allowing for more complicated players. I tackle four significant problems. First, existing commercial RTS games are not suitable for research, therefore I develop a new real-time strategy game as a platform for my work. LagoonCraft, the game I developed, consists of many interoperating systems making up over 300,000 lines of code. Second, I formulate the problem of playing an RTS game as solving a sequence of spatially resolved resource allocation problems. Third, I create a new representation that takes advantage of this problem formulation in order to provide an encoding of RTS game-playing strategy amiable to evolution. Last, I develop a new
method based on co-evolution that generates competent RTS game-playing strategies that routinely beat human opponents.
## Contents

Abstract ................................................................. ii
List of Figures .......................................................... vi

1 Introduction ........................................................... 1
   1.1 Structure of this Thesis ........................................ 5

2 Real-Time Strategy Games .......................................... 7
   2.1 Resource Allocation ............................................ 10
   2.2 Spatial Reasoning .............................................. 11
   2.3 Opponent Modeling ............................................. 12
   2.4 Force Composition ............................................. 13

3 Background and Related Work ...................................... 16
   3.1 Genetic Algorithms ............................................ 16
   3.2 Co-Evolution .................................................. 18
   3.3 Niching and Speciation ........................................ 18
      3.3.1 Fitness Sharing .......................................... 19
      3.3.2 Crowding ................................................ 20
   3.4 Traditional Game AI Research ................................ 20
      3.4.1 Minimax .................................................. 21
   3.5 Commercial Real-Time Strategy Game AI ..................... 25

4 LagoonCraft .......................................................... 27

5 Developing an RTS Game-Player .................................... 33
   5.1 Identifying Resources .......................................... 34
   5.2 Spatial Reasoning .............................................. 35
      5.2.1 Influence Maps .......................................... 37
      5.2.2 Influence Map Specifics ................................ 40
      5.2.3 Influence Map Trees .................................... 42
   5.3 Creating Objectives ........................................... 49
   5.4 Objective Chaining ............................................ 50
   5.5 Allocation .................................................... 52

6 CoQuGa ................................................................. 61
   6.1 Fitness .......................................................... 67
      6.1.1 Niching and Fitness Sharing ............................ 67
   6.2 Encoding ........................................................ 69
## 7 Results

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Phase one - Spatial Reasoning</td>
<td>72</td>
</tr>
<tr>
<td>7.2</td>
<td>Phases One Results: Evolving the Attacker</td>
<td>75</td>
</tr>
<tr>
<td>7.3</td>
<td>Results: Evolving the Defender</td>
<td>77</td>
</tr>
<tr>
<td>7.4</td>
<td>Results: Co-evolving Attackers and Defenders</td>
<td>79</td>
</tr>
<tr>
<td>7.5</td>
<td>Phase 2 - RTS Co-Evolution</td>
<td>83</td>
</tr>
<tr>
<td>7.6</td>
<td>Hand-Coded Opponents</td>
<td>84</td>
</tr>
<tr>
<td>7.7</td>
<td>Phase Two Results</td>
<td>86</td>
</tr>
<tr>
<td>7.8</td>
<td>Analysis</td>
<td>89</td>
</tr>
<tr>
<td>7.9</td>
<td>Phase Two Conclusions</td>
<td>90</td>
</tr>
<tr>
<td>7.10</td>
<td>Results of Phase Three - Full RTS Play</td>
<td>93</td>
</tr>
<tr>
<td>7.11</td>
<td>Phase Three Conclusions</td>
<td>97</td>
</tr>
</tbody>
</table>

## 8 Conclusions and Future Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Contributions</td>
<td>99</td>
</tr>
<tr>
<td>8.2</td>
<td>Improvements and Extensions</td>
<td>101</td>
</tr>
<tr>
<td>8.3</td>
<td>Future Work</td>
<td>102</td>
</tr>
</tbody>
</table>

References 104
## List of Figures

2.1 Supreme Commander [1] ................................................. 8  
2.2 Company of Heroes, another RTS game [2] ............................ 9  
2.3 Production Possibilities Frontier ........................................ 11  
2.4 Warcraft 2 - Fog of War [3] .............................................. 12  
2.5 Paper Rock Scissors Dominance - Archers Dominate Infantry, Which Dominates Cavalry, Which Dominates Archers ......................... 14  

3.1 Genetic Algorithm Overview ............................................. 16  
3.2 One Point Crossover .................................................... 17  
3.3 Minimax Tree .......................................................... 22  
3.4 Orts Screen-shot ....................................................... 24  

4.1 LagoonCraft .......................................................... 27  
4.2 LagoonCraft Unit Dominance Graph .................................... 28  
4.3 Buildings in Lagoon .................................................... 30  
4.4 Squads of Boats in LagoonCraft Moving In Formation ............... 31  

5.1 IMAI Main Loop ....................................................... 34  
5.2 Spatial Reasoning Pipeline ............................................. 36  
5.3 Influence Map Displayed as Numbers ................................... 38  
5.4 Influence Maps for Go, see GoBlob [4] .................................. 39  
5.5 Near Friendly Buildings ................................................ 42  
5.6 Influence Map Tree ..................................................... 43  
5.7 Negative Values near Friendly Buildings ............................... 44  
5.8 Near Friendly Buildings + Away Friendly Buildings ................. 45  
5.9 Game Limitation on Building Locations ................................ 46  
5.10 Away Enemy Buildings ............................................... 46  
5.11 Final Building Placement IM .......................................... 47  
5.12 Final Building Placement IM in A More Complex Situation ......... 47  
5.13 My Objectives in Supreme Commander ................................ 49  
5.14 Objective Chaining .................................................... 50  
5.15 Example Unit Effectiveness Matrix .................................... 58  
5.16 Unit Mixed Strategies ................................................ 58  
5.17 Unit Lifespans ........................................................ 60  

6.1 Master and Slave Model ................................................ 62  
6.2 Master Overview ....................................................... 63  
6.3 Slave Overview ....................................................... 65
Chapter 1
Introduction

Computer and video games are becoming increasingly integrated into modern culture, and while traditional games such as checkers and chess have been the focus of serious AI research, modern video games have not [5, 6, 7, 8, 9]. These games are situated in a virtual world, involve a variety of player skills and decision making processes, and provide a fun immersive experience. Developers of computer players (game AI) for commercial video games tend to utilize finite state machines, rule-based systems, or other such knowledge intensive approaches. These approaches work reasonably well, at least until a human player learns their habits and weaknesses, but require significant player and developer resources to create and tune to play competently. Development of game AI therefore suffers from the knowledge acquisition bottleneck well known to AI researchers and common to many real world AI systems. By using evolutionary techniques to create game players I aim to overcome these bottlenecks and produce superior players.

Real-Time Strategy (RTS) games are particularly interesting as they focus player involvement around making long-term strategic decisions, map to many challenging real world problems, and have yet to be the focus of extensive research. Many excellent RTS games exist, including Starcraft, Dawn of War, Supreme Commander, Company
of Heroes, and Age of Empires [10, 11, 1, 2, 12]. In these games, players are put in charge of a military base or colony with buildings, troops, and money under their control. They play by allocating these resources: spending money to produce units and construct buildings, while assigning various tasks to units. Units carry out these orders automatically, and players win the game by destroying opposing players’ units and buildings.

Video games are fundamentally about making decisions and exercising skills. RTS games concentrate player involvement around making high level, long term strategic decisions.

"A good game is a series of interesting decisions. The decisions must be both frequent and meaningful." - Sid Meier, Game Designer

While varying greatly in content and style, a set of game-playing decisions unifies RTS games into a genre. Most of these decisions can be categorized as either resource allocation problems: how much money to invest on improving my economy, what kind of troops to field, or what technological enhancements to research; or as spatial reasoning problems: which parts of the world should I try to control, how should I assault this defensive installation, or how do I outmaneuver my opponent in this battle. By developing players capable of making these challenging and relevant decisions, I can develop systems capable of tackling important real world problems.

Instead of directly programming my favorite strategy into an artificial player, I
use genetic algorithms to find good game-playing strategies. This reduces the expert knowledge required, allowing for the development of more complicated players that would be difficult to tune with traditional methods. RTS games have, by design, a non-linear search space of potential strategies, with players making interesting and complex decisions - many of which have difficult to predict consequences later in the game. To explore this non-linear search space I use co-evolutionary genetic algorithms, which have been historically effective at difficult search problems.

This work descends from my previous work using case-injected genetic algorithms to play one aspect of RTS games, learning from experience to anticipate opponent moves. That work has been published in the IEEE Transactions on Evolutionary Computation [13] and in the IEEE Congress on Evolutionary Computation [14, 15, 16], the IEEE Conference on Computation Intelligence in Games in 2005 [17], the Genetic and Evolutionary Computing Conference [18], and several others [19, 20, 21]. This dissertation extends that work towards evolving complete RTS game-players.

The central claim of this dissertation is that:

*Effective Real-Time Strategy game players can be co-evolved.*

To do this, I developed LagoonCraft, a real-time strategy game combining gameplay elements from several popular commercial games as a test-bed for research. LagoonCraft follows most RTS game-playing paradigms while allowing for faster then real-time evaluations across a large distributed system. To play LagoonCraft, I de-
veloped the Influence Map based Artificial Intelligence or IMAI, a general RTS game-
player that uses a new representation of RTS strategy. The IMAI plays the game
according to a set of parameters which allow the specification of a wide variety of
effective and competent strategies. To make spatial reasoning decisions I develop
Influence Map Trees (IMTrees), a novel spatial reasoning system that provides a rep-
resentation of spatial reasoning strategies. IMTrees are used by the IMAI to analyze
the game-state to find spatial objectives for the IMAI to carry out. These objectives
tell the player where to attack, defend, expand and build. Combining objectives from
the spatial reasoning system with non-spatial objectives, the IMAI creates a directed
graph describing all interrelated objectives the player is working towards. This graph
can then be analyzed, using a combat estimation algorithm to predict the outcomes
of potential battles, to estimate benefits and assign priorities for each of the objec-
tives. The IMAI then searches for the resource-objective allocation that maximizes
the benefit achievable with currently available resources. This allocation can then be
trivially converted to game-playing commands and used to play the game.

I then developed a co-evolutionary GA, which encodes key IMAI parameters
within the individuals of its population, each of which then represents a distinct
game-playing strategy. By playing individuals in the population against one another,
analyzing the results of those games, and using standard genetic operators to create
new individuals I can co-evolve towards increasingly competent game-playing strate-
gies. Results show that co-evolution produces innovative, robust, strategies capable of defeating a suite of hand-coded opponents and humans.

1.1 Structure of this Thesis

The next chapter provides an overview of real-time strategy games, including discussion of their foundational game-playing decisions and how they relate to real world problems. This chapter can be skipped if you have prior game-playing experience.

Chapter 3 provides an overview of genetic algorithms and summaries of academic research in games and commercial RTS AI techniques.

Chapter 4 describes LagoonCraft, the RTS game I developed to facilitate this research. This chapter discusses the units, buildings, and decisions involved in playing LagoonCraft. More information on LagoonCraft is available at http://lagoon.cse.unr.edu, including the complete source code. LagoonCraft was initially described at the Congress on Evolutionary Computation in 2006 [22].

Chapter 5 details the new problem formulation and describes the architecture of our RTS game-player. This chapter provides detailed descriptions of our approaches to spatial reasoning, resource allocation, and combat estimation. These developments have been published at the Congress on Evolutionary Computation in 2006, and the Symposium on Computation Intelligence in Games in years 2006 and 2007 [22, 23, 24].

Chapter 6 describes my co-evolutionary genetic algorithm. I explain the encoding of RTS strategies, as well as the architecture and execution model used by the
distributed co-evolutionary algorithm. This new technique was first published at the Symposium on Computation Intelligence in Games in 2007 [24].

Chapter 7 provides results showing the effectiveness of our approach. Results are divided into three phases, covering the major issues outlined earlier. The last chapter provides conclusions and a discussion of possible directions for future work.
Real-Time Strategy games (RTS) are a genre of computer and video games. They take place in real-time and involve resource gathering, base building, technology development and high-level control over individual units. Within an RTS game, players become the leader of a colony or military base. For example, in Supreme Commander (SupCom), shown in Figure 2.1, the player controls an Armored Command Unit (ACU), which is an enormous robot capable of constructing and controlling an army of other robots via nano-technology. At the beginning of the game several players are teleported into the world, where they begin by constructing a base. RTS games have a strong economic side, with players in SupCom constructing, upgrading and managing a variety of buildings that produce the two basic resources - mass and energy. Players invest these basic resources into improving their economy, fortifying their base, strengthening their military, and developing various forms of technology. SupCom has hundreds of military units encompassing land, sea and air theaters. Once constructed, players maneuver their military units around the world, attacking and engaging enemy units. The ultimate goal being to destroy the ACU and level the base of any opponents foolish enough to challenge your supremacy.

Underneath the surface game-play, video games are fundamentally about making
Figure 2.1  Supreme Commander [1]
decisions and exercising skills. A car racing simulation involves a great deal of skill in controlling the vehicle along with decisions involving the choice and setup of the vehicle. RTS games, while varying greatly in content and style, are unified by a set of common decisions that their players make. These decisions involve a variety of challenging problems that players are simultaneously solving: developing their economy, training and equipping an effective military, estimating the location and composition of enemy forces, predicting incoming attacks or defensive vulnerabilities, while hiding and misleading their enemies about their own intentions. In contrast to other types of games, the decisions involved in RTS games concentrate player involvement around making high level, long term strategic decisions. The paradigm is designed to draw players into the game world giving them a set of interesting decisions.
to make, along with compelling reasons about why they are making them and their resulting consequences. Real-Time strategy games present a unique opportunity for research, containing a variety of interesting research problems within an integrated and motivated whole. The next sections detail some of the decisions common to many RTS games - how they are presented by the game-play and how they relate to real world problems.

2.1 Resource Allocation

Resource allocation decisions involve allocating a limited set of resources to a set of objectives or tasks; dividing available funds between economic development and military expansion, distributing military production amongst various types of units, and deploying combat forces along several simultaneous fronts. Resource allocation decisions are extremely common in RTS games, one of the most common being the “Guns versus Butter” decision where players make a trade-off between developing their economy or expanding their military. In economics, this problem is a well studied manifestation of a production possibilities frontier. A production possibilities frontier exists when a producer has to compromise between two goods to produce. In Figure 2.3, for example, the producer is compromising between producing tanks and farms. Producing more tanks comes at a cost of decreased farm production. In RTS games players compromise between reinvesting in their economy and expanding their military. This is a highly complex problem in both the real and virtual worlds. The
player who maintains a stronger economy almost always wins in the long-term, but
decisive strikes by a player with a military advantage can devastate an opponent’s
economy.

2.2 Spatial Reasoning

Spatial reasoning problems involve decisions made about the spatial shape of the
battlefield: where do I attack, where do I defend, which parts of the world should I
try to control, how should I assault this defensive installation, how do I outmaneuver
my opponent. These fundamentally difficult decisions form a significant part of RTS
game-playing strategy and relate to many other problems in computer science, such
as the traveling salesman problem, circuit design, and many problems in the fields of
pattern recognition, signal processing, and computer vision. To make spatial reason-
ing decisions within the realm of RTS games I develop Influence Map Trees (IMTrees) discussed in Section 5.2.3.

2.3 Opponent Modeling

Opponent Modeling problems result from the imperfect information presented by the game since RTS players play with an incomplete picture of the game-state. In contrast to games like chess, and in common with games like poker, RTS games do not present their players with complete knowledge of the game-state. Players can only see enemy units within the vicinity of their own troops, the rest of the world is covered in a “fog of war”. Figure 2.4 is a screen-shot from Warcraft 2 showing

Figure 2.4 Warcraft 2 - Fog of War [3]
the traditional three levels of information. Inside the black area in the upper left, no information is available. If a friendly units passes through the vicinity of that area, then it will be promoted to the next level of information which is shown as the gray layer. Inside this level the terrain has been revealed but enemy units are concealed. The lighter areas, around the shipyard and the lumber-mill are currently visible, with any enemy units in those areas revealed. The fog of war obscures enemy units or buildings within it, allowing players to deceive and mislead one another. Developing systems capable of deceiving, misleading, and anticipating one another has been the subject of significant research interest, particularly within the game of poker [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35].

2.4 Force Composition

Force composition decisions involve players determining how to allocate their military budget amongst the various types of troops they can build. For example, suppose a player in SupCom is planning an attack against an opponent early in the game. There are four different tier one ground units available, determining how many of each to bring to the attack is a force composition decision. Players win battles in RTS games by bringing a superior mix of units to the fight. Force composition is less about numbers and more about counter units and cohesive strategies. RTS games are designed to not have dominating strategies, they enforce this by having units follow a paper-rock-scissors model. No single type of unit can defeat all other types of units,
each has its own strengths and weaknesses against the various types of adversaries. In a medieval combat game for instance a player might have three units: archers, infantry, and cavalry. In this game archers defeat infantry, cavalry defeat archers, and infantry defeat cavalry. This leads to a paper-rock-scissors model as shown in Figure 2.5.

Well made games do this in interesting ways: infantry dominate archers or cavalry in close combat, but archers quickly reduce them from range before they can engage, while cavalry can move quickly enough to avoid significant losses to the ranged attacks of archers. Force composition decisions quickly become intertwined with spatial reasoning decisions. Placing infantry in front of archers, for example, protects them from opposing cavalry, forcing opponents into difficult compromises. A modern commercial RTS game like Medieval Total War 2 might have hundreds of types of units.
organized into dozens of categories with many complex interactions, all of which pro-
vide opportunities for players to utilize. Determining the proper composition of units
is a complex and challenging problem, I approach this problem through an objective
chaining and combat estimation system described in Sections 5.4 and 5.5 respectively.

There has been some research in developing RTS game-players. Before describing
my approach, the next chapter introduces related work within the realms of academic
and industrial AI.
Chapter 3
Background and Related Work

This chapter first details genetic algorithms and co-evolution, techniques used heavily in this work. It then explores previous work in RTS games, including traditional game AI research and commercial AI techniques.

3.1 Genetic Algorithms

Genetic Algorithms (GA’s) originated from studies of cellular automata conducted by John Holland and his colleagues at the University of Michigan [36]. They are adaptive methods based on the genetic processes of biological organisms which may be used to solve search and optimization problems. A genetic algorithm is an it-

Figure 3.1 Genetic Algorithm Overview
erative process containing a population of potential solutions. Each individual in
the population encodes a candidate solution to the problem, usually as a bit-string -
1101101001100001111100100000, which in this case maps to a candidate allocation.
Figure 3.1 outlines a genetic algorithm. A fitness function evaluates individuals, and
based upon this fitness, individuals are recombined and manipulated by the genetic
operators of selection, crossover and mutation, to create new individuals. Selection
probabilistically emphasizes the survival and reproduction of higher fitness individu-
als. Crossover combines and exchanges information between individuals while muta-
tion tweaks and optimizes solutions over time.

Selection creates a new generation by iteratively selecting members of the previous
population to reproduce, for example canonical roulette wheel selection picks indi-
viduals with probability proportional to their fitness. Crossover occurs on the newly

![Figure 3.2 One Point Crossover](image)

generated population. One-point crossover picks pairs of individuals and randomly
chooses a location in the bit-string, swapping bits on one side of the divide as shown in
Figure 3.2. Mutation makes small changes to the occasional individual in the popula-
tion. Bit-wise mutation randomly flips bits with some low probability. The resulting search behavior is effective across a range of difficult search problem, with the GA making effective compromises between intelligently exploring new possibilities while exploiting previous good solutions [36].

3.2 Co-Evolution

A canonical genetic algorithm evaluates individuals in isolation, relying on an objective fitness function to measure an individuals utility. Many problems exist, particularly within games, where it is difficult to judge the quality of an individual without comparing it against other individuals. If Bob the chess player asks you how good of a player he is, it is difficult to assign him a rating without having him play a game against an opponent. Co-evolutionary Genetic Algorithms are GA’s where the fitness of an individual is evaluated with respect to other members of the population. In the past co-evolution has been successful in games such as chess and checkers [6] and should be equally applicable within the realm of RTS games.

3.3 Niching and Speciation

Often it is advantageous for the population to contain a set of individuals representing a variety of good solutions, instead of converging on the single best solution. Most interesting games exhibit paper-rock-scissors dynamics, where no single strategy dominates all other strategies. Consider the case where Strategy A defeats strategies
B through J, while strategy K defeats only strategies A and D. While strategy A would be considered the best single strategy, defeating virtually all opponents, it still has a valid counter strategy and the final population should contain individuals A, K, and the best counter strategy for K. A traditional GA would assign higher fitness to strategy A, and the population would eventually converge on that strategy. To prevent this, several techniques have been developed to help maintain diversity. I next describe two such niching techniques.

3.3.1 Fitness Sharing

Fitness sharing is the best known and most widely used niching technique. It was originally introduced by Holland [36] and improved by Goldberg and Richardson [37]. Fitness sharing modifies the search landscape by reducing the payoff in densely populated regions, lowering each individual’s fitness proportional to the number of similar individuals in the population. Generally the fitness is divided by the niche count, determined by a problem specific “niche” function which determines the number of similar individuals in the population.

\[
\text{adjustedFitness} = \frac{\text{fitness}}{\text{nicheCount}}
\]  

(3.1)

In my example, suppose 27 copies of strategy A exist in the population, 4 copies of strategy E exists, and 2 copies of strategy K exist. Each individual receives fitness equal to the number of other strategies they defeat, divided by the niche count.
Strategy $A$ individuals receive $9/27$ fitness, strategy $E$ receives $1/4$ and strategy $K$ receives $2/2$. Strategy $K$ has the highest fitness, and will likely gain a larger share of the population in the next generation. The system should converge on a ratio of $9^{th}$ population $A$, $1^{th}$ population $E$, and $1^{th}$ population $K$. Fitness sharing is effective, but creating the fitness niching function can be difficult as it requires a measure of similarity between players. I use a form of fitness sharing developed in section 6.1 to maintain niching within my population.

### 3.3.2 Crowding

An alternative scheme is DeJong’s crowding [38], where only a small portion of the population reproduces every generation, and offspring replace the most similar individual in a random sample of the population. This makes it difficult for a single strategy to dominate the population since children of good individuals tend to replace each other instead of replacing different strategies. Crowding does not tend to develop the kind of stable populations that fitness sharing does, and it relies upon a “crowding factor” coefficient that is difficult to set. I do not use crowding in this work.

### 3.4 Traditional Game AI Research

A large body of work exists in which evolutionary methods have been applied to games [6, 39, 8, 40, 7]. However the majority of this work has been applied to board, card, and other well defined games. Such games have many differences from popular
real time strategy (RTS) games such as Starcraft, Total Annihilation, Homeworld or Dawn of War [10, 41, 42, 11]. Many traditional board, card and papers games use entities or pieces that have a limited space of positions (such as on a board) and restricted sets of actions (well defined movement). Players in these games have well defined roles, and the information available to players is well defined. These characteristics make the game state easier to specify and analyze. In contrast, entities in RTS games exist and interact over time in continuous three dimensional space. Players do not directly control entities, but instead provide them with higher level goals that lower level AI controllers work to accomplish. For example, players do not tell their units exactly how to maneuver or which specific enemy unit to fire upon with a specific weapon. Instead players give high level commands such as move to this area, fighting everyone you see. This adds a level of abstraction not found in traditional games. These issues push RTS games into a category separate from traditional board games, requiring new algorithms and techniques. John Laird [43, 44, 45] surveys the state of research in using AI techniques in interactive computer games, describing the importance of such research while providing a taxonomy of games and their related problems.

3.4.1 Minimax

One of the keystone techniques in traditional game AI, minimax or minmax is a fundamental tree search algorithm used to play many games. Minimax is a method
in decision theory for minimizing the maximum possible loss. Alternatively, it can be
tought of as maximizing the minimum gain (maximin). It started from two player
zero-sum game theory, covering both the case where players take alternate moves and
the case where players make simultaneous moves. It has also been extended to more
complex games and to general decision making in the presence of uncertainty. A
minimax algorithm is a recursive algorithm for choosing the next move in an $n$-player
game, usually a two-player game. A value is associated with each position or state of
the game. This value is computed by means of a position evaluation function and it
indicates how good it would be for a player to reach that position. The player then
makes the move that maximizes the minimum value of the position resulting from
the opponent’s possible following moves.

Figure 3.3  Minimax Tree

Figure 3.3 shows an example minimax tree where each node represents a potential
future game-state. Minimax determines all possible future steps up to some number of moves in the future, known as the ply. It starts by using a game-state evaluation function to determine the benefit at each of the terminal nodes. Then it propagates values up the tree to determine benefit for the intermediate nodes. At nodes where the next move is an opponent move it takes the minimum value of that nodes children - assuming the opponent will make the move that gives you the worst possible state. Conversely at nodes where it is the players turn, it takes the maximum of the children values, assuming you will act in your own best interest. The depth to which the tree can be explored determines the intelligence of the player. Deep blue used minimax to defeat Gary Kasparov, the world chess champion, in 1997 using specialized hardware to search between 6-12 ply ahead for most moves, with up to 40 ply deep during the end-game. Given enough computation power the complete tree of possible game-state tree could be created for any deterministic discrete state game from tic-tac-toe, chess, checkers, arimaa to go. Minimax could then be used to completely solve the game: creating a player that could not be improved upon. Chinook, for example, has completely solved checkers for board states with 10 or less pieces [46], a process which took 18 years worth of computation. Other board games such as chess and go have much larger game-states that cannot be completely solved with current hardware.

Several recent research projects have been applied to computer and video games. Soar is a symbolic cognitive architecture, created by John Laird, Allen Newell, and
Paul Rosenbloom at CMU [47]. Soar has been applied to two commercial games, Quake 2 and Descent 3, both of which are first-person shooters. While these kinds of games and their associated decisions differ significantly from RTS games, there are elements of how Soar uses long-term objectives subdivided into more immediate tasks that resemble my objective chaining system.

Orts, a real-time strategy game engine developed at the University of Alberta by Michael Buro and his students, was developed in parallel with my work and has similar goals [48]. The primary Orts game, shown in Figure 3.4 is built around the

![Figure 3.4 Orts Screen-shot](image)

game-playing mechanics used in Starcraft. While several players have been developed using Orts, the work has primarily focused on lower level tactical scenarios, using traditional techniques for strategic play.
3.5 Commercial Real-Time Strategy Game AI

RTS games have game-states that cannot be easily enumerated - units exist in continuous space and time, and both players act simultaneously. Because of this many traditionally successful AI techniques such as Minimax cannot be easily applied. Most commercial RTS games use traditional AI approaches such as decision trees, expert systems, and scripted systems. These systems generally follow an “attack in waves” approach: building up troops for a few minutes, with developers scripting which troops to build; sending those troops off to attack and die, repeating with the next wave until the game is over. Simple systems handle the rest of the AI player’s decisions, managing the economy according to pre-programmed rules and using simple algorithms to determine base layout. For example, Starcraft [10], the best selling RTS game and second best selling PC game so far in history, uses the attack in waves system described by the following table.

<table>
<thead>
<tr>
<th>Wave</th>
<th>Zealots</th>
<th>Dragoons</th>
<th>Archons</th>
<th>Carriers</th>
<th>Arbiter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wave 1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wave 2</td>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wave 3</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wave 4</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wave 5</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
While this may provide a challenging opponent a handful of times, competent players quickly learn dominating counter strategies that allow them to easily defeat computer opponents virtually 100% of the time. This is boring. Developers use these kinds of systems because they are a low risk way to provide specific experiences to players. Developers can be confident that at time T in the game their AI will perform action A, leading player into situation B at which point players can overcome seemingly insurmountable odds by making decision C. While this works fine for long scripted campaigns, in shorter skirmish matches it quickly becomes an exercise in seeing the AI do the same actions time and time again. These systems have a significant cost in expert knowledge as well, requiring extensive tuning and testing to produce even moderately challenging players. Even the best scripted systems are inherently predictable and fragile, with human players quickly learning dominant counter strategies. To counter this, virtually all commercial RTS AI systems cheat to provide a more interesting experience for competent human players - providing the AI with a complete map of the world as well as additional resources. While more modern AIs have made strides to overcome these criticisms, generally by employing subtler cheats, RTS AI systems have yet to approach the realm of human-level competence without cheating.

As a testbed for this kind of research I next describe LagoonCraft, a purpose built RTS designed for serious AI research, not entertainment.
Chapter 4
LagoonCraft

I developed LagoonCraft, a real-time 3D naval combat strategy game, as a platform for this research. While there are many excellent RTS games, they are developed for entertainment and are not an optimal environment for research. LagoonCraft has game-play similar to other RTS games: players gather resources, construct buildings,
create units, and then maneuver those units to conquer the world. LagoonCraft has two types of resources: oil, produced by capturing and holding resource points; and power, generated by constructing power generators. The two fundamentally different modes of generating resources requires players to compromise between fighting to expand and control as many points as possible to maximize oil income while reinvesting as much as possible into power generators to maximize power income. Figure 4.1 shows a screen-shot from LagoonCraft where the blue player is attacking from the top right, pushing a heavy destroyer through red’s lines and flanking around the left side of the screen. The red player struggles to maintain a defensive line, pulling back larger ships to engage the destroyer while smaller boats engage blue’s second attack wave.

Figure 4.2  LagoonCraft Unit Dominance Graph
LagoonCraft has a variety of units organized into three tiers: tier one units are squads of small boats armed with light weapons, tier two units are medium sized frigates, and tier three contains heavy destroyers and cruisers. Units have a complex paper-rock-scissors dominance graph between them, shown in Figure 4.2. Unit to unit dominance is primarily enforced by weapon and armor characteristics: the missiles fired by a hellfire frigate have significantly more penetration than the heavy machine guns on an assault frigate, making them more effective against larger units, however the missiles are not accurate enough to effectively engage smaller boats, leading to paper-rock-scissors subgraphs like the one between the two frigates and the RPG boat in Figure 4.2. Players must carefully choose which types they construct, trying to bring the mix of units that best counters their opponent’s forces. Higher tier units provide more firepower for their cost in oil, but are more expensive in terms of power. This allows players to choose divergent economic strategies: concentrating on capturing oil points to produce lots of lower tier units, or reinvesting in power generators to produce higher tier units.

LagoonCraft has a selection of buildings that players can construct shown in Figure 4.3. Players start the game with a headquarters that constructs scouts and generates a modest oil income. More headquarters can be built, and while prohibitively expensive, the game is lost by losing all of your headquarters. Resources are generated by oil platforms, which are captured but cannot be constructed, and power generators
which are built normally. There are three factory buildings that construct units: a
dock which constructs tier one units, a harbor which constructs tier two units, and a
shipyard which constructs tier three units. The higher tier factories are bigger, more
expensive, and more difficult to destroy.

Instead of having players control every action of every one of their units and build-
ings, LagoonCraft, like all RTS games, uses a hierarchical AI system to distribute the
work. At the highest level, humans or strategic AI’s make broad sweeping decisions
which are passed down to the lower level AI’s. A player in SupCom for example
might grab 200 aircraft and tell them to do a formation move near an enemy town,
and then a patrol over their airspace. Middle level AI systems control units on the
squad level, coordinating groups of units to achieve the tasks handed down by the strategic commander. Those 200 aircraft would be divided into small wings of six to twelve units by the squad AI, with each squad moving in formation to the enemy town. Faster aircraft would slow down, or even swing around to group up with slower aircraft, until enemy units were encountered. The middle level AI system also deals with timing and coordination, breaking long term tasks into smaller ones that can be handled by the unit AI. At the lowest level, the unit AI controls individual units to achieve short term tasks via immediate actions like maneuvering to avoid objects, staying in formation, and maintaining proper firing positions. Once the tasked aircraft engage enemy ground units a general melee occurs, with fighters chasing enemy aircraft and bombers bombing enemy buildings. RTS games do not involve direct
control over a single game entity like most other types of computer and video games, instead players direct artificial agents to carry out tasks and policies.

LagoonCraft is a very large project, with around 300,000 lines of c++ code and nearly 50,000 lines of python code. Once developed it has provided an excellent testbed for my research in developing RTS players. More information on LagoonCraft can be found at http://lagoon.cse.unr.edu. It has been used for a variety of other projects, both at UNR and within the greater community.

The LagoonCraft research testbed allowed the investigation of a new approach to play RTS games. This new approach is detailed in the next chapter.
Chapter 5
Developing an RTS Game-Player

I seek to design an AI that can make RTS game-playing decisions. Instead of directly hand-coding the decision making process, I aim to develop a more general player that provides an evolvable representation of RTS strategy. Increasing the complexity of strategies contained within the representation allows for potentially better players, it also significantly increases the amount of expert knowledge required to properly tune and develop such players. Co-evolving these strategies addresses my target goals of reducing the significant cost in expert knowledge and the relative fragility of game-playing strategies found in current hand-scripted systems. I therefore developed the Influence Map based Artificial Intelligence, or IMAI, to address these issues and play LagoonCraft. Playing LagoonCraft requires algorithms that can make the variety of game-playing decisions presented by the game, including spatial reasoning, resource allocation and opponent modeling. My goals for the IMAI are to develop a player that plays complex and robust strategies while containing those strategies within a representation amiable to evolution. The IMAI works on the abstract level by casting the play of the game as a resource allocation problem. IMAI players run in the continuous loop shown in Figure 5.1. Each iteration has three major phases: resource identification, objective creation, and resource allocation. In the resource identifica-
tion phase the system analyzes the current game-state determining which resources (money, buildings, units) are available for the player to allocate. In the objective creation phase the IMAI outlines objectives to achieve. Spatial objectives are first created through a spatial reasoning system as described in Section 5.2. The IMAI then creates non-spatial objectives including creating units, constructing buildings, and determining the importance of gathering resources. The interrelationships between objectives form a directed graph which propagates benefit from objectives to their perquisites as described in Section 5.4. This produces a set of objectives with associated benefits. The IMAI then uses an allocation system to search for good allocations that maximize the benefit realizable with currently available resources as described in Section 5.5. Finally, the IMAI converts this allocation into the actual commands used to play the game.

5.1 Identifying Resources

I define a resource as something players utilize to achieve objectives. The IMAI currently identifies the following types of resources:
1. **Basic Resources:** Oil, Power

2. **Builders:** Capable of constructing units

3. **Units:** Capable of moving, fighting and capturing

4. **Build Points:** Places buildings can be constructed

Identifying resources requires the AI to parse the game-state and determine the relevant information. Most of this information is readily provided by the game. The final and most complicated category of resources, build points are unique in that they represent locations on the map to construct buildings, something that requires a spatial reasoning decision to determine. To identify these locations I use a simplification of the spatial reasoning system used to create objectives. I develop this general spatial reasoning system in the next section.

### 5.2 Spatial Reasoning

The spatial reasoning system analyzes the game-state in order to produce a set of spatial objectives for the IMAI to carry out, see Figure 5.2. Since spatial reasoning decisions form much of the core RTS game-play, the IMAI represents its spatial reasoning strategies in a way amiable to evolution. While spatial reasoning is an area of significant research in computer science and powerful techniques such as qualitative constraint calculi [49] exist, I develop a novel spatial reasoning system - IMTrees - to represent spatial reasoning decisions in an evolvable way.
An enormous variety of spatial reasoning strategies exist in RTS games - even among seemingly simple decisions like where to place basic buildings complex strategies emerge. For example, Starcraft players can use basic supply depots to funnel enemy melee units into well defended kill zones. Representing these strategies in a generic and expressive way while maintaining enough focus to be evolvable is one of the cornerstone problems tackled in this work. Inspired by a wide variety of sources, I use a tree structured representation forming simple building blocks into a larger structure to contain complex ideas within a simple representation. I use Influence Maps (IMs) as a basic spatial reasoning block to contain a simple spatial concept such as ”away from enemy factories”. By forming these blocks into an Influence Map Tree or IMTree I develop a representation capable of containing complex spatial reasoning strategies. IMTrees have similarities to Genetic Programming, an outgrowth of genetic algorithms that evolves tree structured programming elements to solve a variety of problems [50]. While IMTree based spatial reasoning has significant differences, the successes and failures of genetic programming highlight many of the
strengths and weakness involved in evolving tree structured representations. In the next two sections I describe influence maps and influence map trees, the foundations of my spatial reasoning system.

5.2.1 Influence Maps

An influence map (IM) is a grid placed over the world, with values assigned to each square by some function (IMFunction). Once calculated, each influence map relates some spatial feature or concept determined by the IMFunction. Influence maps evolved out of work done on spatial reasoning within the game of Go and have been used sporadically since then in various video games [51]. Figure 5.3 is a visualization of an influence map, with the IMFunction being the number of triangles within some radius. If each triangle was a resource location, and the radius of the circle was the effective resource gathering distance, this IM could be used to find optimal resource gathering locations for any situation.

Influence maps in video games developed to meet the needs of turn based wargames, predecessors of real-time Strategy games, that placed a high priority on competent AI. Influence maps are still used in a variety of games, especially within the RTS genre. For example, Age of Empires [12] uses influence maps to determine building locations (as part of its attack in waves system) using an IMFunction based on nearby related resources. For example, to construct a granary, which stores food, an IM summing up the number of food producing entities within the effective radius of
that building is created. By finding locations in the IM with high values, the AI can determine good locations to place its buildings. Influence maps can be rendered in a graphical format that humans can read fairly easily - Figure 5.4 is displaying various characteristics of a Go board [4]. Because of their readability influence maps can be developed, tested and debugged in ways not possible with other spatial reasoning approaches such as neural networks.

IMs have traditionally been hand-coded to solve particular problems, with a set of IMs combined through a weighted sum to produce solutions to more complicated problems. In contrast, I use a tree structure to allow for a more expressive and general representation.
Figure 5.4  Influence Maps for Go, see GoBlob [4]
5.2.2 Influence Map Specifics

I use one fundamental type of influence map, the “near” IM, which has values based on the vicinity or distance from a type of unit or building. To completely define an IM we need to specify a number of parameters and coefficients, listed below:

Parameters

1. **Relevant side** encoded as a two bit binary integer, is one of the following:
   
   (a) Friendly units
   
   (b) Enemy units
   
   (c) Neutral units
   
   (d) Any side

2. **Relevant Entity Category** encoded as a two bit binary integer, is one of the following

   (a) Units
   
   (b) Buildings
   
   (c) Resource points

3. **Effective Radius** encoded as a two bit binary integer, is one of the following:

   (a) 1000 meters
(b) Effective weapon range of relevant units

(c) \(10 \text{ meters } \times \text{ Unit Size}\)

4. **How values are distributed within the circle** encoded as a two bit binary integer specifying one of the following types of distributions:

   (a) Highest values near the unit, linearly fall off towards zero at the radius

   (b) Highest value along the circumference, linearly falling off towards the unit

   (c) Consistent values throughout the circle

   (d) Effect only the single closest square

5. **What values to assign** encoded as a two bit binary integer specifying one of the following values or sources:

   (a) One

   (b) Value of relevant entities

   (c) Power of relevant entities, where power is an abstract value assigned to the entity denoting tier

   (d) \(\frac{\text{Power}}{\text{Value}}\) of relevant entities

**Coefficients**

1. **Radius Coefficient** which is multiplied by the effective radius and ranges between 0 and 4
2. **Value Coefficient** which is multiplied by the value within each square of the IM, and ranges between \(-10\) and 10

For example, in Figure 5.5 there is a single friendly headquarters, which is adding value to the squares surrounding it based on the value of the headquarters. Values linearly fall off as you move away from the headquarters up to a maximum radius of 1000 meters multiplied by a radius coefficient of approximately 1.3.

![Figure 5.5 Near Friendly Buildings](image)

5.2.3 **Influence Map Trees**

I contain IMs within a tree structure instead of the traditional weighted list [51] to allow increased complexity while retaining the same basic components. Each tree represents a complete decision making strategy in a form that can be encoded and
Figure 5.6  Influence Map Tree

evolved by a genetic algorithm. Leaf nodes in the tree are regular IMs, using an IMFunction to generate their values based on the game-state. Branch nodes perform operations on their children’s values in order to create their own values. These operations include simple arithmetic operators to form new values, such as combining their children’s values in a weighted sum or multiplication to form new values. These nodes also perform processing on their children, smoothing or normalizing their values. Many game AI developers use specialized post-processing to manipulate and customize their influence maps. For example, Age of Empires [12], uses multi-pass smoothing on influence maps to determine locations on which to construct buildings. By allowing nodes in my tree to perform such processing, a single IMTree can concisely represent a large variety of spatial reasoning strategies.
An example of an IMTree is shown in Figure 5.6. This IMTree is an evolved strategy determining where the player should place buildings. The first leaf IM is a "near friendly buildings" node, shown in Figure 5.5, which tells the player to build buildings in the general area of existing ones. The second leaf IM is a "negative values near friendly buildings" node, shown in Figure 5.7, which tells the player not to build too close to existing buildings. The second IM differs from the first in that its radius coefficient is smaller, and its value coefficient is negative, leading it to have high negative values within a smaller radius. These two nodes are children of a summation branch node, which sums them up to produce the IM shown in Figure 5.8. This IM is then multiplied by two further IMs to produce a resulting IM that represents the complete IMTree. The first multiplier is a game determined limitation on how
far away players can construct their buildings, shown in Figure 5.9. The building limiter restricts players to build things near other buildings, and not on top of land. The second multiplier is an “away enemy buildings” node, which generally biases the player to build as far from enemies as possible, shown in Figure 5.10. The final result is the IM shown in Figure 5.11. This IMTree leads the player to construct buildings in a rough hexagon grid, which is an efficient pattern. Note that this IMTree encodes a spatial reasoning strategy that can be applied to any situation, later in the game when more buildings have been constructed the IMTree produces a resultant IM that looks like Figure 5.12.

The IMAI possesses the following IMTrees, each of which has a specific purpose:

1. **attack** details places to send units to attack
Figure 5.9  Game Limitation on Building Locations

Figure 5.10  Away Enemy Buildings
Figure 5.11  Final Building Placement IM

Figure 5.12  Final Building Placement IM in A More Complex Situation
2. **defend** details places to send units to defend

3. **stage** details places to send units to move

4. **civilian construction** details where to construct power generators

5. **military construction** details where to produce factories

Each of these IMTrees reduces to a single influence map, with high values near promising locations to carry out the IMTree task. The IMAI then uses a local maxima finding algorithm to convert this IM into a set of objectives that resources can be allocated to. The algorithm works by iteratively creating objectives at the highest point in the influence map that is not within some distance of other objectives. These objectives receive a raw benefit score based on the value of the IM at that point, this raw benefit scores is used by the allocation system described in Section 5.5.

The IMAI uses three basic types of branch nodes, an operator node which performs mathematical operators on multiple children, a gaussian node, that applies a gaussian blur function to a single child, and a normalization node that normalizes a single child. The operator nodes expose a single parameter to the encoding, describing which operation it performs on their children. The operation can be either:

1. Addition, value for a square is the summed values for that square in all children

2. Subtraction, the first IM is added with negated versions of other children
3. Multiplication, child values are multiplied together

4. Division, the first child value is divided by the other children

5. Maximization, the highest value out of all child is taken for a square

6. Minimization, the lowest value out of all children is taken for a square

Normalization nodes always normalize between 0 and 1, and gaussian nodes expose a single coefficient describing the radius of the gaussian blur.

5.3 Creating Objectives

Figure 5.13  My Objectives in Supreme Commander

An objective is a task which the player considers beneficial to accomplish. The IMTree system first creates spatial objectives to determine where to attack, defend,
and expand. The IMAI then creates construction and training objectives for each type of building and unit. Finally, the IMAI creates resource gathering objectives which determine the value of the various resource types to the player. Having described the spatial reasoning system, the following sections detail objective chaining and resource gathering.

5.4 Objective Chaining

Many objectives do not have self determined benefit, instead being beneficial with respect to reaching other objectives. For example, the benefit associated with capturing points depends on both the spatial location of the point which determines how quickly can it be captured and how easily can it be defended, as well as the overall situation of the player’s economy, which determines how badly the player needs the resources that point produces. Objective chaining works by forming the various

![Objective Chaining Diagram](image)
objectives into a directed graph with associated objectives passing benefit to each other via objective links. The IMAI uses three types of links: additive, multiplicative and hypothetical. Spatial objectives are chained to unit creation objectives through hypothetical links - by performing a hypothetical allocation with an addition unit of the type the objective creates, we can take the gain in benefit as the benefit to associate with the unit creation objective. For example, if having a scout allowed me to get 15 additional benefit, I would associate 15 benefit with training more scouts. Unit creation objectives are chained to building construction objectives by simple additive links - if a cruiser is worth 45 points to me and a destroyer is worth 65, then a shipyard, which produces cruisers and destroyers, is worth 110 (unless I already have a shipyard, in which case this is reduced depending on how much the current shipyard is being used). Resource gathering objectives have additive links to create power generator objectives, and multiplicative links to capture points objectives, which come with a normalized fitness from the spatial system relating to how defensible of a location they are in. In Figure 5.14 the acquire oil objective is passing benefit to the capture point objectives, which in turn passes benefit to the create scout objective, as that is the best unit to capture points. This benefit finally chains on to the create docks objective, leading the player to assign benefit to creating docks proportional to its need for resources - which is very high at the beginning of the game because income is very small. The objective chaining system propagates
benefit from objectives to their prerequisites with the result being a collection of objectives, from places on the map to attack and defend, to construction and unit creation orders. Each objective has an associated benefit, determined both from its own merit and its relationship to other objectives.

5.5 Allocation

Once resources have been identified and objectives have been created, the IMAI searches to find a good allocation of resources to objectives. The purpose of allocation is to find the resource to objective allocation that maximizes expected benefit. To determine this expected benefit, I create resource-objective benefit functions which estimate the benefit the player will receive by allocating some set of resources to an objective. Most objective types have raw benefit scores associated with them as they are created, this benefit being partially realized by the set of resources allocated to that objective. Benefit functions return benefit for unfeasible allocations: allocations which cannot achieve the objective still have some estimated benefit - allocating 2/3 of the money required to construct a building returns 2/3 of the raw benefit associated with constructing that building. Allowing benefit for unfeasible allocations provides a simpler landscape for the allocation search algorithm. To make sure that final allocations are feasible I also develop feasibility functions, which final allocations are required to pass. I develop three core benefit functions, for the three categories of objectives: spatial, building construction and unit creation, and resource gathering.
The IMTree system assigns a raw benefit to spatial objectives as they are created, which is realized by allocating units, allocating a group of units close to the goal and powerful enough to overcome potential resistance returns maximal benefit, while allocating a smaller group of poorly matched units that are farther away returns less benefit. To determine how well matched units are I develop a combat estimation function which estimates the ratio of strengths between two group of units as described in Section 5.5.

Spatial objectives are feasible if at least one unit has been allocated and the ratio of strengths between friendly and enemy units surpasses a certain threshold. I encode and evolve this parameter to allow the search to coordinate risk-aversion with the player’s overall strategy. Construction and training objectives receive their benefit from the objective chaining system, which propagates it from associated spatial objectives - an attack objectives adding benefit to objectives that train offensive units. For buildings, this benefit is adjusted by a metric of long term usage calculated based on total usage over the last few minutes of each category of building, additional docks are only constructed if existing ones are running at capacity. Construction and training objectives are feasible if enough basic resources have been allocated to purchase the unit and if a builder or build-point has been allocated. Resource gathering objectives have benefit proportional to $C_1/income \times C_2/available$. $C_1$ and $C_2$ are basic real valued coefficients describing the players spending behavior that are encoded and
evolved. The IMAI assigns less priority to stockpiling a resource as the stock or income of that resource grows, leading that resource to be more freely spent. The IMAI does not directly assign resources to achieve resource gathering objectives, using them instead as proxies in the objective chaining system to pass benefit onto the objectives that produce resources: capturing points and building power generators - Section 5.4. The benefit associated with resource gathering objectives is also applied as a cost to allocations of such basic resources, the expected benefit resulting from allocating 1000 resources to Task A is reduced by the value of those 1000 resources. This gives the IMAI a tendency to maintain a reserve of basic resources until high benefit objectives become available (attacking an enemy weakness, or defending against incoming attacks).

To perform the actual allocation I have developed two techniques: a genetic algorithm, and a greedy search algorithm. I first developed a genetic algorithm (AllocGA) to do the resource to objective allocation. AllocGA came from work I did for my masters thesis that used genetic algorithms to do strike force asset allocation within a real-time strategy game [20, 14, 18, 21, 19]. Section 3.1 provides an overview of genetic algorithms. AllocGA is a GA which uses single point crossover, bitwise mutation and roulette wheel selection. AllocGA used a simple enumeration encoding for the allocation, with each resource receiving a set of bits to encode an integer identifying which resource it was allocated to. It encodes potential allocations between resources
and objectives as individuals in its population. The fitness for each individual is a summation of the benefits expected from each element in its allocation. AllocGA was highly effective at finding good allocations, particularly in light of the complexity of some of the allocations. However, AllocGA suffered from two fundamental issues: it was computationally expensive to run a GA while the game was running, and when rerun it would produce allocations that were similar in overall benefit but different in implementation. The second issue led to a critical case of indecisiveness as it would frequently redirect boats to other goals, to the point where nothing was accomplished. For example, the player has two scouts and two points it wants to capture on opposite sides of the map. The first run through AllocGA sends scout 1 to point A, and scout 2 to point B. Thirty seconds later when run again, it redirects scout 1 to point B, and scout 2 to point A. It repeats this process indefinitely, with neither point ever being captured because the scouts are constantly turning around. To solve this problem I developed a greedy algorithm to perform the allocation. The greedy algorithm works by scanning all possible resource and objective pairs and taking the one which provides the most benefit. It then repeats on the remaining resources and objectives so long as its continues to increase the sum benefit of the complete allocation. The greedy allocator was significantly faster and more repeatable then the GA, but was not as good at finding good allocations in the face of more complicated situations, such as when it takes a particular mix of units to effectively assault an
Combat Estimation

I develop a combat estimation system that the IMAI uses to estimate the benefit it will receive from sending its forces to engage enemy units. This is a non-trivial and important task for RTS game players to do which requires significant experience for humans to learn. This is also an area where commercial RTS AI is traditionally weak, forming one of the most effective ways for humans to manipulate artificial opponents. To do this correctly I need a function that properly estimates the outcome of combat between any two collections of units. I first outline several desired properties for this function:

1. More > Less, 5 Riflemen should defeat 4 Riflemen

2. Environment - Skirmishers in broken terrain defeat Cavalry but Cavalry in open terrain defeat Skirmishers.

3. Mixed Forces - 5 Riflemen, 3 Anti Tank Guns, 3 Tanks are better than 8 Riflemen, 6 Tanks

4. Heavy weapons - 1 tank defeats 100 riflemen, i.e. some units cannot be defeated purely through numbers

5. Economies of scale - 1 zealot defeats 4 marines, but 40 marines defeats 10 zealots
6. Group Dynamics - Unit types C and D are weak individually, but powerful when used together.

This is a complex problem - units have a wide variety of abilities and skills that cannot be distilled to a single number. Units have many interactions both with supporting units, enemy abilities, and the nature of the terrain itself. Because of this, a perfect estimation of effectiveness would be extremely difficult to abstract out of most RTS games.

Most commercial RTS AI reduces each unit to a single strength value, incorporates some basic bonuses / penalties based on paper-rock-scissors dynamics, and then compares the summed strengths of both groups of units. The systems succeed at reaching the first two properties, but leave significant room for improvement with the last four issues.

I developed a novel combat estimation technique inspired by game-theory. I define the output of the function as a single floating point value between 0 and 1 relating the ratio of strengths between the two groups. A score of 1 means that my forces should completely destroy the enemy with virtually no casualties, while a score of 0 is total defeat, and a .5 is an even fight. I ignore the concepts of terrain and economies of scale, as neither of those are fundamentally important in LagoonCraft. I create a unit-effectiveness matrix describing how quickly a unit can kill any other opposing unit under ideal conditions, an example of which is shown in Figure 5.15. In my example
matrix, for example, the infantry unit can destroy .1 other infantry units within some arbitrary time unit. So in 10 time units a squad of infantry can completely destroy an enemy squad of infantry, while taking virtually forever to destroy an enemy tank.

For each unit on both sides I calculate a mixed strategy for how they will distribute their offensive capabilities. I assume that units can perfectly distribute their fire in whichever way they want. Units distribute their offensive capabilities against the set of enemy units proportional to their unit effectiveness against that unit divided by
their sum effectiveness against all enemy units, as in Equation 5.1.

\[ O(A, B) = \frac{E(A, B)}{\sum_{B' \in N} E(A, B')} \]  

(5.1)

Where \( A \) and \( B \) are units from sides \( M \) and \( N \) respectively. \( O(A, B) \) is the percentage of \( A \)'s attacks that are going to \( b \) and \( E(A, B) \) is the unit effectiveness score describing how much damage \( A \) can do to \( B \). In Figure 5.16 for example, the riflemen on the left are distributing their fire evenly against the riflemen on the right, because they have very little effectiveness against the tank. The tank on the right is effective against most unit types, so it is distributing its fire fairly evenly. The anti-tank guns on the left are primarily useful against enemy armor, so they are concentrating on the enemy tanks.

Once the mixed strategies for each unit have been calculated, we can determine total lifespan for each unit, per Equation 5.2.

\[ \text{Lifespan}(A) = \frac{1}{\sum_{B' \in N} E(B', A)} \]  

(5.2)

For our example this produces the lifespans shown in Figure 5.17.

I then calculate the ratio of strength between the two forces as:

\[ \text{Ratio}(E, F) = \frac{\text{Max}_F(\text{Lifespan}(A))}{\text{Max}_E(\text{Lifespan}(B))/2} \]  

(5.3)

The longest surviving unit on the left is the Tank, which survives 1.85 time units, compared to the tanks on the right which survive 1.38 time units. This gives a strength
ratio of \((1.85/1.38)/2 = .67\). Our estimation system is giving the left group of units a moderate advantage. While the actual outcome of the fight would depend upon many factors, my estimation system does a significantly better job of dealing with the important dynamics of the problem that the “sum and compare” systems present in most RTS games. Expanding my force composition system to deal with terrain could be done by having various unit-effectiveness matrix’s for each type of terrain. Dealing with economies of scale and group dynamics would be very interesting avenues for future research.

These systems combine together to form the IMAI, which can represent and play robust RTS strategies. Instead of hand-tuning these strategies, which is time-consuming and error-prone, I use automated search techniques to generate them. Specifically, I use a co-evolutionary genetic algorithm, as described in the next chapter.
Chapter 6
CoQuGa

With the IMAI providing a representation of game-playing strategies, I can investigate methods for generating effective strategies. I develop a Co-evolutionary Queue based Genetic Algorithm (CoQuGa) to perform this search. Searching to find good strategies instead of hand-coding them reduces the cost in expert knowledge and allows for the development of more complex players. RTS games are designed to have a non-linear search space of complex strategies, making it difficult to apply many traditional search techniques. I use genetic algorithms because they have been historically effective at solving difficult search problems [36].

CoQuGa maintains a population of individuals, each of which in turn encodes a strategy for the IMAI to use. I evaluate these Individuals within CoQuGa by playing them against other members of the population, making CoQuGa a co-evolutionary system. CoQuGa is a steady state GA - frequently creating and destroying a handful of individuals instead of occasionally replacing the entire population. Evaluating individuals requires running the game, which is computationally expensive. To overcome this we use a master-slave distributed model that distributes evaluations over a network of computers, shown in Figure 6.1. Within the master-slave model, the master maintains the population and applies genetic operators while the slaves perform the
actual evaluations. At any given time there is one master process running, with a large number of slaves continuously connecting and requesting work. The master is an event driven system, performing work only in response to requests by the slaves. This produces what we refer to as a “Queue” based genetic algorithm, which maintains a Queue of evaluations for slaves to perform. As slaves take pieces of work out of the Queue, the master generates new work for them, which occasionally involves the creation of new individuals and the application of genetic operators. This is in contrast to a canonical GA that directly iterates through generations. Eliminating the concept of generations reduces the synchronization burden between master and slave, allowing for more efficient networking. Networking is important as evaluating a pair of individuals requires running a complete game, which takes upwards of 10 minutes on a modern machine. Running a large number of evaluations in parallel
CoQuGa uses a master-slave networking model, with the CoQuGa Master, shown in Figure 6.2, responsible for maintaining the population and applying genetic operators. It has no concept of the problem it is working on, instead using a representation template provided by the slaves to construct the initial population. Slaves connect to the master and request work, triggering the master to update the population in order to provide them with an evaluation to carry out. The master’s first order of business is to acquire a representation template, which determines the encoding being used, to create the initial population. The master does this by requesting such a template from any idle slaves, which then send back a set of static opponents, which are hand-coded.
solutions used to determine the layout of the genotype. Once the master receives the representation template it creates the initial population, taking the structure from the first static opponent but filling the bit-string with random bits. When the initial population is created the data is randomized, but under the structure supplied by the first static opponent. These static opponents are also used to judge the objective progress of co-evolution, with a batch of five individuals being sampled out of the population every 25 co-evolutionary evaluations, to be tested against the static opponents. Ideally, as co-evolution progresses players in the population should do increasingly well against the hand-coded static opponents. Note, these evaluations are purely for the purpose of analyzing the progress of co-evolution, they do not affect the fitness of individuals in the population. Once static opponents have been acquired and the initial population has been created, the master finds evaluations for slaves to perform by probabilistically sampling pairs of individuals from the population. The master selects individuals for evaluation via a roulette wheel selection, with probability inversely proportional to the number of evaluations the player already has. These pairs of individuals are transmitted to the slave that requests work, who evaluates them, and then returns the results of the battle. These results contain the scores accumulated for the two players during the game and a judgment of who was the eventual winner.

Individuals are not eligible for reproduction or replacement until they have par-
ticipated in at least five battles. This was empirically chosen as enough of a history to allow for effective analysis. If every member of the population has received enough evaluations, then the master creates new individuals to evaluate. The master creates new individuals by using roulette wheel selection to find a pair of high fitness individuals and a pair of low fitness individuals, via fitness proportional and inverse fitness proportional selection respectively. The master replaces the low-fitness individuals with children of the high-fitness individuals, applying genetic operators to create the new children. Standard one-point crossover and simple point mutation is done on the children. Crossover occurs with 75% probability, and mutation occurs with a probability chosen to flip 2 bits on average per individual.

![Figure 6.3 Slave Overview](image)
CoQuGa slaves, shown in Figure 6.3 are responsible for the problem specific aspects of the Genetic Algorithm. These include converting the hand-coded strategy phenotypes into encoded genotypes, decoding evolved genotypes into IMAI strategy phenotypes, and evaluating competing individuals. As slaves are asynchronously created they connect to the master, requesting work when they are idle. The Master responds with a set of individuals for the slave to evaluate. Upon receiving the pair of individuals the slave resets the world to its initial state, assigning each player a side. These players then play the game, with the slave occasionally checking to see if the game has finished. Games run for 15 minutes of simulation time, 10 minutes during earlier work, at which time the winner is chosen as the player who had the largest sum income of resources at the end of the match. Note, this is not the largest amount of resource available to the player at that time, but instead an accumulation of all the money made during the match. For example, making 3000 oil and spending 2900 of it is preferential to making 2000 oil and only spending 300 of it. Resources gathered is an accurate estimation of eventual victory for most RTS games, and it appears to be accurate within LagoonCraft as well. Many games end quickly however, with one player rapidly dominating the other. Once the game has finished the slave returns the outcome to the master, who uses it to calculate fitness.
6.1 Fitness

A canonical Genetic Algorithm evaluates individuals in isolation, for example calling a black-box function on individual 47 which returns a fitness of 5. Within the context of a game it is difficult to objectively analyze the quality of a player without other players to compare against. Ideally, strategies would be evaluated with a system similar to the Elo system used in chess, assigning fitness to a player based on their past history against a variety of opponents \[52\]. Co-evolutionary Genetic Algorithms are GA’s where the fitness of an individual is evaluated with respect to other members of the population. By testing my players against other players in the population, I open the doors for both open-ended evolution and for more robust strategies. In the past co-evolution has been successful in games such as chess and checkers \[6\] and is equally applicable to evolving RTS strategies. CoQuGa evaluates individuals by playing them against each other, calculating fitness based on the historical win-loss records of each player.

6.1.1 Niching and Fitness Sharing

To improve the performance of my co-evolutionary GA, I use a fitness sharing system that analyzes historical win-loss records to calculate fitness. Players are required to play at least 5 games before they can come up for reproduction or replacement. Under the fitness sharing scheme, each player is worth 100 points, which is distributed to any opponents who defeat that player. Figure 6.4 shows a simple example with
Figure 6.4  Fitness Sharing

four players, player A defeats players B and D, while losing to player C. Note that all players lose to at least one opponent, so there are no clearly dominant players. This is common amongst strategies in RTS games, as they are designed to follow basic paper-rock-scissors dynamics similar to those that apply to unit combat. For example, a skirmishing player might defeat a purely defensive player, while losing to a rushing player, who in turns loses to a purely defensive player. As no single strategy dominates all others, it would be beneficial for the population of evolving players to contain a diverse selection of players. This allows the developer to select from a set of good evolved strategies, and helps ensure robustness amongst the ultimate strategies, so that they have to evolve in the presence of their counter strategies. In Figure 6.4 player C dominates player D by defeating every opponent that D can beat, this leads
D to have a low fitness, making it a candidate for replacement. Fitness sharing rewards players for beating opponents that few other players have beaten, encouraging the population to maintain different game-playing strategies, this ensures that no single strategy becomes overspecialized as it will be tested against a wide cross-section of opposing strategies.

\[ F_n = 100 \times 3^{\frac{\bar{f} - f}{\sigma}} \]  

(6.1)

Equation 6.1 details how fitness is normalized over the population, where \( f \) is the original fitness calculated through fitness sharing, \( \bar{f} \) is the average fitness within the population and \( \sigma \) is the standard deviation of the population’s fitness scores.

### 6.2 Encoding

An IMAI player contains an RTS game-playing strategy within the various parameters and coefficients controlling how it plays the game. By encoding this collection of parameters within the bit-strings of individuals within CoQuGa, I co-evolve RTS game-playing strategies.

The first set of parameters involves the IMTree system. The IMAI has several IMTrees, one for attack, defend, and two build IMTrees dictating where to play civilian and military buildings. Each of these trees has a basic structure, which is defined by the hand-coded players and not evolved. Each IM in the tree, which ranges in size from 10-20 IMs, has a number of parameters that control how the IM functions. Section 5.2.2 details all the various parameters, each of which is encoded.
as either a binary fraction, or as a binary integer which maps to an enumeration. For example, leaf "near" nodes can look at either enemy, friendly, or neutral units. This is encoded as an integer choice, ranging from zero to three, which is encoded as two bits in the bit-string. The actual parameters and their encodings are listed in Section 5.2.2. Branch nodes are quite a bit simpler, having the value coefficient common to all influence maps, as well as an enumeration controlling which operation they perform on their children. Once CoQuGa has packed all of the parameters for each IM in each IMTree within the IMAI player into a bit-string, it looks at the non-spatial parameters. These include the unit-effectiveness matrix, which determines how powerful the IMAI thinks each unit is against each type of opponent. This also includes the resource gathering coefficients, which dictates how urgently the player captures resources, and how cautious they are about spending them. The GA packs all the parameters for each IM in the IMTree into a bit-string, with fixed point binary integer encoding for the enumerations and fixed point binary fraction encoding for the real valued parameters and coefficients.

In the next chapter I test the effectiveness of the IMAI and of CoQuGa.
Chapter 7
Results

The development of LagoonCraft, the IMAI, and CoQuGa took place in three major phases. Phase one concentrated on developing and testing the IMTree system. I constructed a basic IMAI player that combined the IMTree system with AllocGA to determine allocations. This early IMAI player was used to play a tactical derivative of LagoonCraft. We then used an early version of CoQuGA to co-evolve these players, with published results showing the development of effective spatial reasoning strategies [22, 23]. The second phase of development transitioned into full LagoonCraft. To deal with the increased complexities involved in playing an RTS game I developed the non-spatial objective system, the objective chaining systems, and a primitive form of the force composition system. Published results show the co-evolution of RTS strategies comparable to their hand-coded counterparts [24]. The final phase focused on refining the LagoonCraft game-play while improving the IMAI. LagoonCraft units had their statistics rebalanced to allow for a more diverse and dynamic style of play. To overcome some of the weaknesses in the phase two system, the force composition system and resource gathering objectives were developed. This in turn lead to further refinements in the objective chaining system. Results from this latest phase show the co-evolution of players that strongly dominate their hand-coded counterparts. The
next sections detail each of these phases in more detail.

7.1 Phase one - Spatial Reasoning

The initial phase of work concentrated on developing the IMTree system, with the goal of developing a general spatial reasoning system. To test IMTrees, I developed a tactical game shown in Figure 7.1. Two small cigarette boats, shown as triangles from this distance, attempt to attack the oil platform, shown as a pentagon, which is being guarded by a destroyer, which is the elongated hexagon. The cigarette boats are fast, maneuverable, and equipped with rocket propelled grenade launchers. Their primary advantage over the defending destroyer is that there are two of them and that they can quickly accelerate, decelerate and turn. The destroyer on the other hand is also quite fast, with a higher top speed than the attacking cigarette boats, but requiring a significant period of time to turn and change speed. The long-range weapons on the destroyer have been replaced with banks of machine guns, requiring the destroyer to chase down the cigarette boats to engage them. While the cigarette boats have slightly more range with their rocket propelled grenades, the destroyer has far more firepower and will quickly win a fight.

This scenario, while relatively simple, was chosen because it requires players to understand the effectiveness of their units with the possibility of evolving coordinated attacks. This is more of a tactical than a strategic scenario in that there are few boats on each side, and no “complex long term” decisions for players to make.
Figure 7.1  Tactical Scenario
To test my IMTree based player, I developed hand-coded players to play both attacker and defender. My hand-coded attacker has a basic attack-distract behavior, with one cigarette boat trying to distract and occupy the destroyer while the other goes for the oil-platform. My basic defender spent most of its effort chasing after the attackers, hoping to cut them off and broadside them with its machine guns. My hand-coded attackers were reasonably effective. They won most of the missions, but often made mistakes trying to circle around the destroyer to reach the other side.

With the hand-coded players developed, I then used a genetic algorithm to try to evolve superior players. To evolve my players, I used first a non-co-evolutionary version of CoQuGa which calculated fitness by playing individuals against a static opponent. Fitness is calculated as:

\[ \text{fitness} = \text{damage}_{\text{done}} - \text{damage}_{\text{received}} \]  

(7.1)

I first evolved the attacking boats, using my hand-coded defender as an opponent. This produced more reliable attackers that produced less mistakes. I then evolved the defender against the attackers I had just evolved. This produced more cautious defenders that were difficult to lure away from the oil platform. While effective against one another, both of these strategies where fragile against human opponents, showing that the two individuals had over-specialized against one another.

I used co-evolution to improve on this, extending the GA to evolve two populations simultaneously. To perform an evaluation, an attacking player was pulled from the
attacker population and a defender from the defender population. The players would play against each other as before, with fitness calculated according to Equation 7.1. Using this system to co-evolve players lead to robust and effective players for both sides.

The next sections describe these steps in greater detail.

7.2 Phases One Results: Evolving the Attacker

The first experiments were in evolving an offensive strategy for Scenario 7.1. An early version of CoQuGa was used to evolve IMAI players comprised of only the IMTree and AllocGA subsystems. I created a population of 100 random individuals, each of which was tested against the static defender, with fitness assigned according to Equation 7.1. While I ran the system multiple times, I will discuss a single representative run that illustrates the results I consistently achieved. We graph the fitness of the best, worst, and average individual in the population after each evaluation in Figure 7.2.

At first, the randomly created attacker population is full of bad strategies that either drive in circles, directly charge the destroyer, or sail off over the sunset. Eventually, strategies that charge the platforms crossover with strategies that avoid the destroyer to produce attack-avoid strategies, with the attackers trying to avoid the destroyer while getting close to the oil platform. After several hundred more evaluations this evolves into an attack-distract strategy, shown in Figure 7.3, where one attacker
Figure 7.2  Min/Max/Avg of Attackers Evolving Against a Hand-coded Defender

distracts the defender while the other attacks. Unlike my hand-coded attacker, where one boat attacked while the other distracted, the evolved attackers would often switch roles, with one boat distracting for several seconds before returning to the oil platform. The evolved behavior was also more cautious about approaching the destroyer then the hand-coded player, going well out of their way to avoid it. This reduced the problem of my hand-coded attacker skimming across the destroyers weapons range. At first glance the attackers seem very chaotic, frequently changing heading and speed. As a human defending against the evolved attackers I noticed they seem fairly chaotic, frequently changing directions. This makes their paths difficult to predict, which in turn allows them lots of time to attack the oil platform while the defender circles around. Overall their behavior was significantly superior to the hand-coded
attacker.

![Image of evolved attacker behavior](image)

Figure 7.3 Behavior Exhibited by Evolved Attacker

7.3 Results: Evolving the Defender

The evolved attackers were effective against the hand-coded defender, coordinating an effective attack-distract strategy. I next used the genetic algorithm to evolve the defender, with the goal of finding an effective counter to the attackers strategy. The fitness of individuals in the population are shown in Figure 7.4. The attack distract
Figure 7.4  Min/Max/Avg Fitness of a Defender Evolving against the Evolved Attacker

behavior capitalizes well on the advantage given to the attackers, making it difficult for the destroyer to effectively defend. The defenders evolved surpassed the quality of my own hand-coded defenders, learning how to trick the attackers into making mistakes. Figure 7.5 shows an exceptional defense, where the defender pushes both attackers back without them getting even a single shot off at the platform. In most games however, the evolved defenders played similarly to my hand-coded player, trying to chase off the attackers if they get to close to the oil platform without getting drawn too far away. I then played a few games against both evolved attacker and defender, and noticed that while the attacker and defender play effectively against each other, they are still making obvious mistakes against human opponents. The evolved players were highly specialized towards fighting each other, which made them very fragile against
other strategies. To improve on this robustness I utilized co-evolution, aiming to generate more robust players.

7.4 Results: Co-evolving Attackers and Defenders

To implement co-evolution I expanded the genetic algorithm to include two populations, one containing attackers and one containing defenders. Evaluations took place by sampling an individual from each population, and then playing those individuals against one another. Each individual played exactly one match before having fitness assigned, with that fitness based on Equation 7.1. Once again the players are playing the tactical mission shown in Figure 7.1. I allowed each GA to evaluate 1000 candidate strategies from each population, graphing the minimum, maximum, and
Figure 7.6  Min/Max/Avg Fitness's of Attacker / Defender Co-evolving
average fitness in the populations over time in Figure 7.6. At first there is chaos, with both players using random strategies. Effective attackers and defenders emerge however, with the attackers learning to go for the oil platform while the defender learns to go for the attackers. The attackers suffer for a few hundred generations while they learn how to attack the oil platform and avoid the destroyer. Eventually, attack-avoid strategies and attack distract strategies emerge and the attackers fitness rises dramatically. This leads to improvements in the defender, learning not to be lured away from the oil platform, and to keep its speed up. Ultimately behaviors similar to those shown in Figure 7.7 develop, with attackers employing a well rounded attack-distract-avoid behavior while the defender tends to concentrate on circling the oil platform, chasing the attackers only when in a good firing position. Instead of concentrating on an attack-distract behavior like my hand-coded attacker, or the evolved attacker, the co-evolved attackers focused more on an attack-avoid behavior. Co-evolved attackers tended to cut across to the opposite side of the oil platform that the defender was on, fire a few rounds, and then maneuver again to avoid the attacker. The co-evolved defender was less concerned with chasing the attackers than with staying near the oil-platform, attempting to catch the attackers only if a good situation presented itself. This was in contrast to the hand-coded and evolved behavior of chasing them all over the place. As a human playing against the co-evolved players I noticed their strategies were much more robust. Whatever I did as the de-
Figure 7.7  Final Co-Evolved Players
fender the co-evolved attackers would zip around to the far side of me and continue firing. Catching them required anticipating the next location they would maneuver to, and then determining a course that would allow me to maintain my moment as I chased them. This was very difficult, with the co-evolved attackers beating me in most of the games. The co-evolved defender was also quite good, circling the oil-platform to maintain momentum instead of parking next to it like my hand-coded defender. When the attacking boats got too close and it had a good speed advantage it would dart out and destroy them.

Since both attacking and defending strategies had to be effective against many opposing strategies to be maintained in the population, they were more robust than the players which evolved to beat a single opponent or those I hand-tuned. Overall, both co-evolved attacker and defender played robust and effective strategies. While we concentrated on a tactical scenario, results showed the evolution of effective spatial reasoning strategies. Encouraged by these results we expanded the system and applied it to a more strategic game. This is detailed in the next section.

7.5 Phase 2 - RTS Co-Evolution

Results from phase two show that IMTrees can be co-evolved to find effective spatial reasoning strategies. For phase two we built on the IMTree system to construct the IMAI, using it to play LagoonCraft the RTS. The complexity of the game increased dramatically between the tactical and strategic versions. To deal with this I developed
a non-spatial reasoning system, the objective chaining system, a primitive form of the force composition system and the greedy allocator. The non-spatial reasoning system, which manages the building constructing and unit creation objectives, was created to allow the IMAI player to construct building and create units. The objective-chaining system was developed to provide benefit for these non-spatial objectives by linking them in with the spatial reasoning system. The primitive force composition system was used to determine which units to build and allocate to various tasks. In the early force composition system each unit was reduced to three key statistics which were compared against values attached to the IMTree. The result was a simple sum and compare system, with no understanding of the underlying paper-rock-scissors dynamics. These systems allowed the IMAI to play the game with some competence. To test our ability to co-evolve players I then developed CoQuGa, with the goal of co-evolving superior players.

I develop the LagoonCraft map shown in Figure 7.8 for my players to play in. The map is symmetric, with players starting in opposing corners, and resource points overlaid with white circles. Players played 10 minute matches, with fitness calculated as described in Section 6.1.

7.6 Hand-Coded Opponents

I developed three hand-coded opponents against which to test my AIs. Each opponent plays a different but effective strategy. The first player plays a solid rushing
strategy, constructing a basic manufacturing building and then training large numbers of tier one combat ships. It attacks in force early in the game, trying to destroy the enemy’s base before they can prepare. The second player plays a solid defensive strategy, expanding out quickly early in the game, and then concentrating most of its units on defending its points. It quickly builds up to powerful units, only attacking when an overwhelming force is available. The third player plays a skirmish oriented game, distributing its forces to continuously assault on multiple fronts, trying to deny the opponent access to resources. It will aggressively pursue its advantage if it gets momentum, pushing back its opponent’s lines until it reaches their town.

Testing between the three hand-coded players showed the skirmishing player was the most effective overall, defeating the defending player about 70% of the time, and the
rushing player around 45% of the time. The defending player defeats the rushing player about 85% of the time. The rushing player can defeat the skirmishing player by pushing a strong force of units into its town at the beginning of the town. If the initial push fails the skirmishers will quickly capture most of the points on the map, leading to almost certain loss for the rushing player. The defender does not attack aggressively in the late game, which helps preserve its units but makes it difficult to win on resources at the 10 minute mark. Defender against defender battles usually resulted in stalemates, with both sides massing large armies they were unwilling to commit to battle. Stale-mates are common between humans in many RTS games.

### 7.7 Phase Two Results

I create a population of 25 random individuals, which are tested against each other and evolved as described in Chapter 6. After every 25 evaluations, 5 players are sampled at random from the population to play against each of the hand-coded opponents. The IMAI players are not rewarded or punished for winning or losing against the hand-coded players, it is purely used as a benchmark to see how the population is evolving over time. I graph the total score my evolved players receive against the static AIs in Figure 7.9.

Looking at the graph of scores shows improvement throughout the course of co-evolution with players getting increasingly better. The players score the best against the aggressive opponent, which is reasonable because the attacker generally sacrifices
Figure 7.9  Scores Received by Co-Evolved Players Against Hand-Coded AIs
expansion and capturing points in order to attack its enemy’s town. So long as the IMAI players keep their town well defended, they should be able to capture most of the map and score unnaturally high. The balanced player by contrast fights feverishly over every last point, making it difficult for players to accumulate quite so many resources.

Ultimately, the score is just an approximation for how well they are playing, whether they are actually winning is the important matter. I graph average win/loss rates over the number of evaluations in Figure 7.10.

The IMAI players quickly became superior to the hand-coded players, ultimately beating them at around a 72% rate.

Figure 7.10   Ratio of Wins against Static Opponents
7.8 Analysis

Analysis shows that most good IMAI players play strategies similar to the skirmishing player, only with superior coefficients controlling how they balanced their allocations. Co-evolved players tend to send waves of troops in to attack an opponent’s economy, as opposed to the skirmishing AI’s tendency to target manufacturing facilities. An example is shown in Figure 4.1 where an evolved player is flanking its opponent’s line to push in and disrupt its points.

![Figure 7.11: Universe of IMAI Strategies](image)

After watching a large set of random strategies play, it became apparent that skirmishing strategies occupied a significant proportion of the space of possible strategies provided by our representation. It requires only a few properly set parameters in the IMTree to create an effective skirmishing player. The defensive and rushing play-
ers in contrast require several inter-operating parameters. Since skirmishing strategy occupy a large part of the space, as shown in Figure 7.11, the initial GA population usually contains several of them. These strategies are effective leading them to quickly dominate the population. Once the basic strategy contained within the population has converged on skirmishing, the GA focuses on optimizing to find the best skirmisher.

7.9 Phase Two Conclusions

Co-evolving IMAI players produced RTS players superior to their hand-coded opponents. Against the defensive and rushing players the IMAI has some initial problems, because those strategy are underrepresented within the GA population. This keeps evolving players from being exposed to them, leaving them vulnerable to their style of play. As evolution progresses through players become increasingly robust, and these hand-coded strategies become less effective.

The overall win-rates quickly converge at around 72%. After that, scores continued to increase and play became more robust but the overall win-rates were relatively stagnant. This was primarily due to the random sampling used to determine which individuals played against the static opponents. In phase three we use fitness proportional sampling which to test good players from our population against the static opponents. This reduced the number of poor, newly created individuals that were tested against the static players, leading to a significant improvement in win percent-
Because the map was fairly small, and there were only a handful of force-composition parameters to optimize there was a limitation on how much improvement could be done over the initial hand-coded players. This lead to rapid co-evolution of players that won most of the games, but failed to dominate their opponents. Virtually all games were decided by resources at the ten minute mark, as few players could effectively assault their opponent’s base. There was a tendency for evolved players to concentrate their spending very heavily on improving their economy, building an excess of power generators and headquarters in order to improve their resource income. This would put them at a military disadvantage, but since the game would be decided before their opponent could really press home that advantage, it allowed them to accumulate more points. This was a sub-optimal solution, where our fitness function was not effective estimating who would eventually win the game.

I also noted that while over the long term the win/loss ratio was a slow increase, over shorter periods of time the population cycled through a variety of strategies. Players that use small boats are better against the rushing and defending players, while players that use large boats are better against the skirmishing player. Graphing wins averaged over shorter periods of time shows this cycling effect, shown in Figure 7.12. IMAI players tend to cycle through the types of units they prefer during evolution. In the beginning most players use small units, players that use larger boats
are more effective and start to dominate the population. This continues until most players concentrate hard on the largest boats. However, the largest boats can be overwhelmed with a large number of the smaller boats, so the cycle eventually circles back around to smaller boats. While fitness sharing protects the various species, the population size is too small and the number of evaluations too low to reach a stable equilibrium. Since the hand-coded players are static, during times when the IMAI players use the appropriate counter units they win a large percentage of the games. Conversely, during times when they use the wrong units they are less effective.

To improve upon our results, we continued this work into phase three with the goal of creating ever better players.
7.10 Results of Phase Three - Full RTS Play

While phase two co-evolved reasonably effective RTS game-players, room for improvement existed. The primary issue was a poor understanding of the unit domination graph resulting in the use of only a handful of unit types. Efficiently destroying buildings in LagoonCraft, and many other RTS games, requires specialized “siege” units armed with heavy weapons, such as the “Galactic Colossus” in Supreme Commander shown in Figure 7.13. Because both the hand-coded and evolved players failed to understand this, they had difficulty conducting base assaults. This was further exacerbated by the balancing of units in LagoonCraft, and the way the low-level unit controllers were implemented. The net effect was that the defender had a large advantage over attacking troops, making it difficult to destroy someone’s town. This

![A Galactic Colossus Siege Unit from Supreme Commander](image)

**Figure 7.13** A Galactic Colossus Siege Unit from Supreme Commander [1]
led to evolved players that focused on maximizing their economic income instead of trying to destroy their opponents outright. While economics are a fundamental part of the game, the ultimate goal is to destroy your opponent’s base, an act the IMAI rarely followed through on.

To improve on this, phase three saw adjustments in the way LagoonCraft operates along with further development of the IMAI. I developed the force-composition system to provide the IMAI with an improved understanding of how units interrelate. By better understanding the roles of various units, players were able to more effectively attack and defend. I also developed the resource gathering objectives and expanded the objective chaining system to allow for better management of oil and power. This allowed players to save up more money, which made it easier for players to balance the construction of units across all three tiers.

With improvements made to LagoonCraft and the IMAI, I then updated the hand-coded players to create rushing, skirmishing and defensive players. This new generation of players played significantly better, and in contrast to their predecessors often won by elimination instead of always relying on the point comparison. To further encourage this, I lengthened the amount of time given to each evaluation to 15 minutes.

I then re-ran the experiments from phase 2 with the new IMAI and the new hand-coded players. These players play a version of the mission from before that has been
scaled up to 150% of its previous size. As before, the IMAI creates a population of 25 random individuals which are tested against each other and evolved as described in Chapter 6. After every 25 evaluations, 5 players are sampled at random from the population to play against each of the hand-coded opponents. The IMAI players are not rewarded or punished for winning or losing against the hand-coded players, it is purely used as a benchmark to see how the population is evolving over time. I graph the total score received by my evolved players against the hand-coded AIs in Figure 7.14.

Figure 7.14  Phase Three, Scores Received by Co-Evolved players against Hand-Coded AIs
The graph of scores shows a solid improvement throughout most of the course of co-evolution, followed by a decrease near the end. This decrease resulted from an increase in the number of games that ended early, as players defeated their opponents faster they had less opportunity to accumulate points. The defensive player now seems to be the weakest, having benefited the least from the phase three upgrades. As before, I graph the ratio of wins and losses averaged over 25 tests, shown in Figure 7.15.

\textbf{Figure 7.15}  \hspace{1em}  \text{Phase Three, Win Loss Ratio of Co-Evolved Players Against Hand-Coded AIs}

Here we see strong improvement throughout co-evolution, with the later co-evolved
players dominating the hand-coded players. While in phase two, co-evolution took relatively little time to produce good strategies, here we see longer more continued progress. This likely results from the significant increase in information encoded within each player, primarily from the addition of the unit-effectiveness matrix which nearly doubled the amount of information encoded in each individual. With more information the quality of a random individual is lower, but the amount of room evolution has to improve over the hand-coded players increases. I lost several games against the evolved players, with games generally breaking down into three phases. In the first part of the game, the AI used light units to squabble over resources points. After a few minutes they would transition into tier two units good against tier one units, pushing me back into my town. Phase two would be several large waves of units optimized to destroy whatever troops I had defending in my town. Once those units were depleted, the AI would shift to heavier siege units, destroying my buildings rapidly while their remaining forces from tier two kept me occupied. While the evolved players are not invincible, as they lack the tactical understanding to avoid being rushed out at the very beginning of the game, they are a definite improvement over the players from phase two.

7.11 Phase Three Conclusions

Co-evolution lead to IMAI players that dominated their hand-coded opponents. As players continued to evolve they became increasingly more aggressive. This is
comparable with many commercial RTS games such as Starcraft [10], where new players play hour long matches while many professional level games are resolved within ten to twenty minutes.

Phase three players completely dominated phase two players, presenting them with effective offenses that phase two players had difficulty responding too. The evolved players style of combat is very effective against humans. Humans have a difficult time fighting five battles at once, leading the co-evolved players to defeat most human opponents. In play against the other people in my lab, the evolved players were very effective, winning handily the first few times people played against them.
Chapter 8
Conclusions and Future Work

I tackled four significant problems in this work. First, as existing commercial RTS games are not suitable for research I developed LagoonCraft, a new real-time strategy game to fit my purposes. Secondly, I formulated the problem of playing an RTS game as solving a sequence of spatially resolved resource allocation problems. Third, I created a new representation based on IMTrees, a novel spatial reasoning system, to take advantage of this problem formulation in order to provide an encoding of RTS game-playing strategy amiable to evolution. Last, I developed a new method that uses co-evolution to generate competent RTS game-playing strategies. Results show that co-evolution is capable of evolving our RTS game-playing strategies. Our co-evolutionary system is capable, with very little expert knowledge, of developing players significantly superior to hand-coded opponents.

8.1 Contributions

Approaching RTS game-play as a resource allocation problem allowed the development of an architecture that was flexible, modular and effective. Many other situations exist, both within games and many real world problems that could be effectively abstracted into resource allocation problems. Our resource-objective architecture could be readily applied to many of these problems.
Results show the IMT tree system is a representation of spatial-reasoning strategies that can be evolved to perform complex spatial reasoning tasks. Spatial reasoning is an aspect of computer science that has yet to be fully explored, and our system outlines directions for approaching a host of previously difficult real-world problems through evolutionary search techniques.

The objective chaining system is an important step towards developing systems capable of managing multiple short term objectives aimed at accomplishing long term goals. Merging these two together to perform temporal reasoning is another pathway for significant computer science research. Our work developed systems that could be applied across a range of temporal problems within the real world. Within RTS games specifically, the objective chaining system allows for the development of players with a deeper understanding of the interrelations between the various tasks they are carrying out. Many commercial RTS game-players are only effective so long as they can stick to their predefined game-plan. Because of its deeper understanding, the IMAI is capable of dealing with unforeseen setbacks, circumstances, and opportunities. For example, suppose an opponent destroys a tier two factory as the player is preparing for an attack involving tier two units. Most traditional AI’s would delay the attack until the factory can be reconstructed and the desired tier two units built. The IMAI, in contrast, is more flexible in its strategy because it actually understands the role those tier two units play. If they are truly necessary, the IMAI will rebuild the tier
two factory and attack. Often times however, other units can work as a substitute in the interim, something a more traditional system would be unaware of. The objective chaining system allows for more robust and effective strategies. It could be applied to a variety of real world applications to help deal with unexpected situations.

The force composition system was a significant improvement on existing techniques. While computationally intensive, it provided more accurate estimations of relative strength. This allowed players to effectively use mixed forces, a fundamental weakness in many existing RTS AI systems, and a key element in the push towards human level competence.

Our novel co-evolutionary genetic algorithm, CoQuGa showed its ability to evolve a complex representation across a large unreliable heterogeneous network. In spite of continuously crashing slaves and networking issues it reliably produced high quality results. Co-evolution proved itself by producing superior players where were robust against a variety of opponents. Informal matches between co-evolved players and humans showed that the evolved players were very competent, defeating everyone I could convince to play against them.

### 8.2 Improvements and Extensions

The one system of the IMAI that has need of improvement is the allocation system. While both the AllocGA and the greedy algorithm find effective allocations, they are computationally expensive. This was one of the fundamental findings of previous
work, that we confirmed within RTS games [20, 14, 18, 21, 19].

Because we approach the game as a sequence of independent allocation problems, it is difficult to implement a concept such as retreat. A more dynamic system that tied objectives together over time would allow the IMAI to analyze how well its allocated resources were achieving their tasks. This would better allow units to retreat or be reallocated to more urgent tasks.

There are two common aspects of RTS games the IMAI has not been extended to deal with. First, are technological upgrades, where players make a large investment to purchase an upgrade that lasts for the rest of the game. This could be implemented fairly easily as another category of objectives in the graph if a function describing benefit could be determined. Research is not present in all RTS games, but is common enough to warrant being implemented. The second element is defensive buildings. Most RTS games allow players to construct towers, turrets, walls or even moats and complex traps to defend their town without having to actively use units. There was some promising preliminary work along these lines within LagoonCraft, that would be interesting to explore further.

8.3 Future Work

There are many directions for future research that would be interesting to explore. First, to experiment with testing co-evolved players against human opponents, determining if they are truly approaching human level competence by playing against each
other. There are issues in this, such as determining the competence of the human opponent, or training players before they engage the computer. Another important direction for future work is to test our system directly against other RTS AI systems. We are attempting to start a LagoonCraft contest at the Conference on Computational Intelligence in Games in 2008, with the goal of having other players entered by 2009. This would allow more objective tests between systems, to determine how effectively we actually are playing the game. Testing our system against commercial AI systems would also be very enlightening. Only a handful of games provide the interfaces required to do this, but the number of games supporting these kinds of mods are ever increasing. Playing a major commercial game that cost millions dollars to develop would better illustrate the strengths and weakness of our techniques.

Finally, we would like to apply some of these techniques to related real world problems. IMTrees can be co-evolved to make spatial reasoning strategies thus it would be interesting to use them in real world problems such as predicting mineral locations.
References


