Case-Injected Genetic Algorithms in Computer Strategy Games

by

Chris Miles

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

Reno, Nevada

May, 2006
We recommend that the thesis prepared under our supervision by

**CHRISTOPHER EOIN MILES**

entitled

*Case-Injected Genetic Algorithms in Computer Strategy Games*

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

**MASTER OF SCIENCE**

Sushil Louis, Ph.D., Advisor

Monica Nicolescu, Ph.D., Committee Member

Thomas Quint, Ph.D., Graduate School Representative

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

May, 2006
ABSTRACT

Case-Injected Genetic Algorithms in Computer Strategy Games

by

Chris Miles

We use case injected genetic algorithms to play computer strategy games involving complex long range planning with imperfect knowledge of the game state. The dynamic nature of these games requires players to anticipate opponent moves and adapt their strategies accordingly. We use genetic algorithm to play these games, casting them as a resource allocation problem, solutions of which map to effective gameplaying strategies. Results show this is effective with the genetic algorithm searching towards near optimal game-playing strategies. We then develop a learning technique, constructing a case-base of information which can be used to anticipate opponent moves. Methods are developed for the acquisition and elicitation of this knowledge both from past play, and from the observation of human experts. Results show the genetic algorithm produces near-optimal strategies that accomplish the mission while anticipating and avoiding opponent moves.
Acknowledgments

I would like to thank my wife Jigna for the countless reasons that make her the awesome Jigna that she is. I would like to thank my mom too for still being nice after putting up with me for all these years. I would also like to thank Professor Louis and all the gaslab people who were involved in the work and made it as fun as it has been - Anil, Kai, Ryan, Adam, Juan, David. I would also like to thank all the trees that gave there lives to support the printing of this work, without their sacrifice it would not have been possible.

This material is based upon work supported by the Office of Naval Research under contract number N00014-03-1-0104.
Contents

Abstract iii
Acknowledgments iv
List of Figures vii

1 Introduction 1
1.1 Real Time Strategy Games 3
1.2 Strike Ops and GAP 5
1.3 Structure of this Thesis 8

2 Previous Work 10
2.1 Genetic Algorithms 10
2.2 Case Injection 13
2.3 Game AI 14

3 Strike Ops 17
3.1 Sequence of Play 18
3.2 Popups and Anticipation 18
3.3 Summary 20

4 GAP - The Genetic Algorithm Player 22
4.1 Encoding 24
4.2 Routing 26
4.3 General Routing Knowledge 29
4.4 Fitness 31
4.4.1 Probabilistic Health Metrics 32
4.5 Results - GAP Can Play the Game 34

5 Dynamism and Re-planning 37
5.1 Case Injection for Re-planning 40
5.2 Results 40
5.2.1 Game Complexity 41
5.2.2 Re-planning Scope 44

6 Anticipation and Learning 50
6.1 Reflecting on Past Games 52
6.2 The Scenario 53
6.3 Results - Learning From Experience 55
6.4 Learning from Others 56
# List of Figures

1.1 RTS Games Left: Dawn of War [1], Right: Starcraft [2] . . . . . . . . 3
1.2 Screenshots of GAP playing Strike Ops . . . . . . . . . . . . . . . . . 6

2.1 Genetic Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 One Point Crossover . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Case-Injection Stores Information Across Similar Problems . . . . . . 14
2.4 Continually Injecting and Extracting Individuals . . . . . . . . . . . . 14

3.1 Game Screenshots . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Gameplay overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.1 GAP’s Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Allocation Encoding . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.3 How Routes are Built From an Encoding . . . . . . . . . . . . . . . . 26
4.4 A Pathfinding Graph . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.5 Left: $RC < 1$, Middle: $RC = 1$, Right: $RC > 1$ . . . . . . . . . . . . 31
4.6 The Mission . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.7 Best/Worst/Average Individual Fitness as a function of Generation -
Averaged over 50 runs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

5.1 Considering the continuous game as a discrete series of game states . 38
5.2 Yellow path was re-planned when the pop-up occurred . . . . . . . . 38
5.3 Case-injection for Re-planning . . . . . . . . . . . . . . . . . . . . . . 41
5.4 Mission Complexity versus Evaluations Required - Left and Fitness -
Right Top: Complex - Middle: Moderate - Bottom: Simple . . . . . . . . 43
5.5 Simple mission re-planning scenario . . . . . . . . . . . . . . . . . . 45
5.6 Moderate mission re-plan scenario . . . . . . . . . . . . . . . . . . . . 45
5.7 Complex mission re-plan scenario . . . . . . . . . . . . . . . . . . . . 46
5.8 Re-planning Size versus Evaluations Required - Left and Fitness - Right 48

6.1 Reflection Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 53
6.2 Left - The Trapping Scenario, Right - Routes . . . . . . . . . . . . . . 54
6.3 Histogram of Routing Parameters produced without Case Injection. . 56
6.4 Left: Effect of Case Injection on Fitness Inside the GA over time Right:
Effect of Population Size and the Number of Generations on Percentage
Green routes Produced . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.5 Left: Learning From Others Mission - Right: Reverse Engineered Plans 60
6.6 Histogram of Routing Parameters produced without Case Injection. . 61
6.7 Histogram of Routing Parameters produced with Case Injection . . . 62
6.8 Alternate Mission . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.9 Histogram of Routing Parameters produced without Case Injection on the Alternate Mission. . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.10 Histogram of Routing Parameters produced with Case Injection on the Alternate Mission. . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.11 Number of Evaluations effect on the Percentage Green routes Produced 64
6.12 Left: Without Case Injection - Middle: With Case Injection - Right: With Fitness Inflation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Chapter 1
Introduction

Computer games are becoming increasingly integrated into modern culture, and while traditional games such as checkers and chess have been the focus of serious research, modern video games have not [3, 4, 5, 6, 7]. These games are situated in a virtual world, involve a variety of player skills and techniques, and provide an immersive, fun experience. Computer games are more than just entertainment as many training, planning, and scientific problems can be formulated as games where user decisions determine the final outcome.

Developers of computer players (game AI) for computer games tend to utilize finite state machines, rule-based systems, or other such knowledge intensive approaches. These approaches work 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.

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

Games are fundamentally about making decisions, many of which can be cast as non-linear optimization problems. A player in a tactical combat game might have
six soldiers in their command, to whom they assign weapons and armor and give objectives to carry out. Decisions interact with each other in many ways, for example sending soldier C in with heavy weapons draws attention which helps soldier F sneak around and knock out the power. Allocating these resources (soldiers, equipment, and objectives in this case) is a complex non-linear optimization problem, based on a set of interdependent decisions with complicated interactions. Thus, decision making in many games can be cast as such optimization problems. Genetic algorithms, a search technique inspired from evolution, were designed to solve such poorly understood problems.

The central claim of this thesis is that

*Case-Injected Genetic Algorithms can play computer strategy games, learning from experience to anticipate opponent moves.*

Our research is focused primarily on computer strategy games, in particular Real Time Strategy (RTS) games. These are games such as Starcraft, Dawn of War, Age of Empires, or Homeworld [2, 1, 8, 9]. Examples are shown in Figure 1.1. These games are fundamentally resource allocation problems, a class of problems on which genetic algorithms have been historically effective [10]. While varying greatly in content and play, they share central underlying decisions involving resource allocation, spatial reasoning and opponent anticipation which can be readily mapped to real world applications.
1.1 Real Time Strategy Games

Players in real time strategy games generally possess a set of abstract resources (gold, crystal, mana, saltpeter), as well as units and/or buildings. Allocating these resources forms the core gameplay. Units can be allocated to gather resources, or to attack and defend various parts of the map. Simultaneously, abstract resources are being used to produce, maintain, and upgrade units and buildings.

Games in this genre vary greatly in their particulars: setting, environment, specific strategies, but are unified by a set of central underlying decisions. One example of a central underlying decision in these games is the classic "guns and butter" decision, where players must choose between producing more troops or developing a better economy.

This is a highly complex decision; players that go for troops early in the game
can weaken or destroy their opponent quickly before they can develop, but a player with a stronger economy has a significant advantage later in the game. There are many allocation decisions like this, that are in themselves relatively simple, but the decisions all interact with each other leading to complex and interesting gameplay.

Spatial reasoning problems are the second type of decisions in these games. For example, consider a player deciding where to engage their opponent. Open terrain benefits cavalry and armor, while broken terrain benefits skirmishers and light infantry. If your army has superior ranged fighting capabilities, you would want terrain which limits your enemies movement - keeping them from engaging you in close combat. However, if you have greater mobility you would want more open terrain to capitalize on that advantage. Players must look at their knowledge of the terrain and form assumptions about how battles in various areas will take place against their enemies expected forces. Of course the enemy has their own ideas on where to fight, so it might be necessary to lure, bait, or otherwise deceive them.

*All warfare is based on deception. Hence, when able to attack, we must seem unable; when using our forces, we must seem inactive; when we are near, we must make the enemy believe we are far away; when far away, we must make him believe we are near. Hold out baits to entice the enemy. Feign disorder, and crush him.* - Sun Tzu, *the Art of War*

Deception and anticipation are the third category of decisions, and are common
amongst many competitive games. Players in RTS games usually have a limited view of the game-state, they know only the locations of nearby enemies which contrasts with games like chess where the location of all pieces is known. This is part of the imperfect knowledge presented to players. Transitive paper-rock-scissors dominations are common in RTS games: infantry defeat cavalry, which defeat artillery, which defeat infantry. Players who anticipate which units their opponent will field, can counter with the appropriate units and gain an advantage. A player who can anticipate their opponent has an advantage over one who cannot, a player who cannot be anticipated has an advantage over a player who is transparent.

These three categories of decisions form the basis for interesting real time strategy games. Our ultimate goal is to develop players for these games which make intelligent decisions, anticipating and manipulating their opponents in order to win. This thesis develops a genetic algorithm to make allocation decisions within the context of strike ops, an RTS game which focuses its gameplay into allocation decisions. We then develop a genetic algorithm capable of playing a perfect knowledge version of the game. Finally we extend our genetic algorithm to learn and adapt, allowing it to effectively play the game with incomplete knowledge.

1.2 Strike Ops and GAP

We developed Strike Ops, a computer real time strategy game as a platform for research. Figure 1.2 shows some screenshots. Strike Ops is an RTS game correspond-
ing tightly with a real world problem while presenting the foundational decisions common amongst RTS games. To play Strike Ops we developed GAP, the Genetic Algorithm Player. GAP plays by casting the game as a non-linear optimization problem, which GAP solves with a GA - Genetic Algorithm. GAP converts the solution to the optimization problem into a plan of action, which can be used to play the game. To deal with imperfect knowledge and the dynamic nature of the game, GAP re-plans, rerunning the GA when the game-state changes unexpectedly. This is effective, producing near-optimal responses to whatever changes have happened, but it is computationally expensive. To reduce computation time we utilize case-injection, which has been shown to improve search time [11]. Case-Injection works by saving individuals from the population of one run of a GA and injecting them later into
the population of a GA solving a similar problem. We extract individuals from the original plan, and inject them into future re-plans. This allows GAP to maintain knowledge – results show significantly faster productions of plans of equal or better quality. With case-injection GAP can quickly re-plan and respond to changes in the game-state, playing an effective reactive game in the face of imperfect information.

Anticipation and proactive play is superior to reactive play and while being good at getting out of tough situations is useful, but it is ultimately better to avoid them in the first place. To realize this transition we again utilize case-injection. Case-injection has the side effect of biasing search towards injected material, we exploit this side effect by using case-injection to lead the GA towards producing plans with particular traits. By building a case-base of plans anticipating opponent moves and then injecting those plans, we bias GAP to play in an anticipatory manner. We first develop methods for building this case-base through the extraction of knowledge from past experience, leading GAP to improve with each game played. Results show GAP avoid areas in which it has been trapped in the past, anticipating them based upon past experience. Extending this technique, we develop methods for adding to the case-base by eliciting knowledge from other players, particularly human experts. Reverse-engineering their gameplaying strategy into a case allows GAP to learn general lessons from their gameplay. By injecting those cases GAP applies that knowledge, biasing GAP to play more like the human from which it has learned. With case-injection
we are able to produce a player that learns from experience and from other players, significantly improving its gameplay when faced with imperfect knowledge.

The result of all this work is a genetic algorithm player which can efficiently find and play near-optimal plans of attack, it learns from experience what kinds of opponent defenses are likely, anticipating and avoiding them. It can also be biased by the introduction of knowledge from human players, allowing it to play any strategy desired.

1.3 Structure of this Thesis

Chapter 2 describes previous work related to this project, including an overview of genetic algorithms and case-injection. We discuss past work in game AI, research in more traditional games, as well as industry techniques for RTS game AI.

Chapter 3 describes the game of Strike Ops, its fundamental design decisions, and the motivation for those decisions. We also discuss how Strike Ops relates to other real-time strategy games and real world applications.

Chapter 4 describes the development of GAP, our genetic algorithm player, and how it plays Strike Ops. We explain the encoding and evaluation of plans, as well as its connections with more traditional techniques. Results show that GAP can play the game, producing near-optimal plans.

Chapter 5 discusses how GAP deals with the dynamic nature of the game through re-planning. We explore the limitations of that technique, and we use case-injection
to help overcome those limitations. Work published in the Symposium on Computational Intelligence and Games showed that re-planning is effective at producing good plans, and that case-injection provides significant improvements in re-planning speed while maintaining or improving plan quality.

Chapter 6 covers anticipation and learning. We develop methods for using case-injected GA’s to learn from past experience. We explore anticipation in the context of traps, with GAP learning where the defender is likely to have left traps. We develop techniques first for extracting knowledge from GAP’s past experience and then from the play of others. We show how GAP applies acquired knowledge, adapting it to new situations while maintaining important strategic elements. This learning is general across a variety of similar missions, leading to robust play in the face of many situations. This work was first published in the Genetic and Evolutionary Computation Conference [12], showing the GAP can learn from its own past experience, avoiding traps similar to those it has seen before. Later work was published in the Conference on Evolutionary Computation [13], showing GAP can learn from human players - anticipating the same traps those humans were playing in anticipation of.

Chapter 7 summarizes this thesis’s contributions and outlines directions for future work.
Chapter 2
Previous Work

This chapter first overviews the two techniques used heavily in this work - genetic algorithms and case-injection. It then explores previous work in this field, including traditional game AI research and industry techniques.

2.1 Genetic Algorithms

Genetic Algorithms (GAs) originated from the studies of cellular automata conducted by John Holland and his colleagues at the University of Michigan [14]. 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 iterative process containing a population of potential solutions. Each individual in the population encodes a solution to the problem, usually as a bitstring - 1110001010101011010110. Figure 2.1 outlines the genetic algorithm process. A fitness function evaluates individuals, and based upon their fitness individuals are recombined and manipulated by genetic operators to create new individuals and solutions. Genetic operators include: selection, which biases the survival and reproduction of higher fitness individuals; crossover, which combines and exchanges information between individuals; and mutation, which tweaks and optimizes solutions over time.
Consider a genetic algorithm which tries to produce cost-efficient and utilitarian vehicles. The GA would initialize a population of random bitstrings, each of which could be mapped to a car design. The population would contain a set of random cars with various components, properties and characteristics. Each car would be evaluated based upon its utility and its cost, reducing it to a single fitness value. A team of human experts could analyze the designs and assign a value, or a simulation could construct a virtual car and run it through a battery of tests. These utility values could be combined with algorithms that determine the total cost of producing such a car, based upon the cost of various components, and expected labor and machinery. The resultant fitness value would be a good measure of the fitness of that car design, with cheap but useful cars scoring highly. Using this fitness information, the genetic algorithm applies the selection, crossover, and mutation operators to the population to produce a new generation of vehicles. First, selection determines which
individuals survive to the new generation. In roulette wheel selection individuals are chosen to reproduce to the next generation with probability proportional to their fitness compared to the average fitness in the population. Individuals with higher than average fitness reproduce more, crowding out lower fitness individuals. Second, crossover takes individuals that have been chosen to reproduce and recombines their genetic information to produce offspring. In canonical one-point crossover a location is randomly chosen in the bitstring, bits on either side of the divide are swapped as in Figure 2.2. A car and a truck are recombined producing a car with the frame of a truck - an SUV if you will, and a truck with the interior of a car.

![One Point Crossover](image)

**Figure 2.2** One Point Crossover

While crossover works by recombining individuals, mutation works by taking a single individual and applying random changes - tweaking the suspension travel, or exchanging disk brakes for drum brakes. Mutation produces new individuals which are similar to old ones, if the new individuals are better they are more likely to survive, leading to gradual improvement over time. One the genetic operators have produced a new generation of individuals, the process repeats iteratively, until a ”good-enough”
solution has been found or the allocated computational time has been exhausted.

2.2 Case Injection

Case-Injection combines a genetic-algorithm with case-base memory. The intuition is that problems seldom exist in isolation, and a GA is likely to encounter a large number of similar problems over its life-time. By maintaining information learned on similar problems in the past as in figure 2.3, the GA can improve its performance over time.

In the canonical GA, the population is randomly seeded at the start of every problem and destroyed at the end. Figure 2.4 illustrates how in case-injection good solutions (individuals) from each problem are extracted and stored into a case-base, from which they are injected into the population when solving other similar problems. While the GA is running it extracts individuals to the case-base. Individuals who are superior to the previous best are saved into the case-base. This produces a case-base containing a sequence of best individuals in the population. Every few generations in future runs of the GA, individuals are injected from the case-base into the current population. The effect is that of biasing the GA to look towards answers similar to those that were previously successful. If the solution to the problem injected from is similar to the solution to the current problem injection will improve convergence speed and solution quality - Louis[15].

Case-based reasoning research has shown that this question of problem similarity
is non-trivial, case-injection resolves this by probabilistically choosing individuals to injected based upon their similarity (hamming distance) to the current best.

**Figure 2.3** Case-Injection Stores Information Across Similar Problems

![Diagram](image)

**Figure 2.4** Continually Injecting and Extracting Individuals

### 2.3 Game AI

Previous work in strike force asset allocation has been done in optimizing the allocation of assets to targets, the majority of it focusing on static pre-mission planning. Griggs [16] formulated a mixed-integer problem (MIP) to allocate platforms and as-
sets for each objective. The MIP is augmented with a decision tree that determines the best plan based upon weather data. Li [17] converts a nonlinear programming formulation into a MIP problem. Yost [18] provides a survey of the work that has been conducted to address the optimization of strike allocation assets. Both of these techniques worked on the allocation problem alone, developing algorithms to produce asset/target pairings. Our work differs in that it combines the allocation as one part of a larger 3D game, introducing complexity in the form of routing and traps. To deal with this complexity we use more general techniques to search for possible answers instead of trying to directly produce the optimum. Louis [19] applied case injected genetic algorithms to strike force asset allocation, showing results consistent with the effectiveness of our GA.

A large body of work exists in which evolutionary methods have been applied to games [4, 20, 6, 21, 5]. 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[2, 22, 9, 1]. Many traditional (board, card, paper) games use entities (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 also have well defined roles and the domain of knowledge available to each player is clearly identified. These characteristics make the game state easier to specify and analyze.
In contrast, entities in our game exist and interact over time in continuous three dimensional space. Entities are not directly controlled by players but instead sets of algorithms control them in order to meet goals outlined by players. This adds a level of abstraction not found in those traditional games. In most of these computer games, players have incomplete knowledge of the game state, and even this domain of each player’s knowledge is difficult to identify. John Laird [23, 24, 25] surveys the state of research in using Artificial Intelligence (AI) techniques in interactive computers games. He describes the importance of such research and provides a taxonomy of games. Several military simulations share some of our game’s properties [26, 27, 28], these however are military simulations while ours is intended as a platform for research in strategic planning.
Chapter 3
Strike Ops

We developed a computer strategy game, Strike Ops, as a platform for our research. Strike Ops was designed to present the fundamental real-time strategy decisions while having a tight correspondence with a real world application. Strike Ops was also designed to have as few non-strategic decisions as possible, so that it lacks the micromanagement common in many other real-time strategy games. Two opposing and asymmetric sides play Strike Ops: Blue and Red. Figure 3.1 shows the basic elements of the game. Blue plays by sending aircraft (platforms) to attack Red’s buildings (targets) and defensive installations (threats) with various bombs and missiles (assets). The various assets are limited in supply and have varying effectiveness against each target. Because of the scarcity of assets and the potential for well armored targets, Blue has to make a complex decision in allocating its assets to enemy targets. Red primarily plays by placing its defenses (threats) to defend the targets. The different types of defenses have particular effectiveness against the various platforms, along with varying ranges at which they can detect and fire upon the platforms. Both players seek to allocate their respective resources in order to maximize the damage done to their opponent while minimizing the damage taken by their units. The game is dynamic; weather and other environmental factors affect
asset performance, unknown threats can pop-up and new targets can appear to be destroyed.

3.1 Sequence of Play

Figure 3.2 outlines the sequence of action during the gameplay. Both players are presented with the scenario at the beginning of the game and time is given to prepare their initial strategies. The scenario contains information such as the resources available to both units, the location of the targets and the starting location for the platforms. Red first constructs its defense, looking at the layout of the targets, as well as the landscape and the starting location for blue. Blue then constructs its attack plan taking into account both its own resources and the layout of Red’s defenses. Once both players have constructed their plans the game begins. During the game both players can alter their strategy, Blue can reroute or re-prioritize its attackers, and Red can activate / deactivate popups (covered in section 3.2). When any surviving platforms return home the mission concludes and scores are tabulated for both players.

3.2 Popups and Anticipation

Strike Ops includes traps, in the form of pop-ups, as a fundamental part of the gameplay. Radar system can be detected at very long range, much longer than the radar itself can detect. As a result most defenses are known in advance to the attacker.
Figure 3.1  Game Screenshots
The defender can however deactivate its defenses in order to keep the attacker from detecting them. They can be activated later in order to surprise the attacker during a mission. Pop-ups allow a range of strategic options for the defenders. By cleverly locating threats Red can feign vulnerability and lure Blue into a deviously located pop-up trap, or keep Blue from exploiting such a weakness out of fear of a trap. Pop-ups are an important part of the gameplay, and they model both strategy in the real world game as well as a range of decisions in real-time strategy games. Other unexpected events can happen in the game, but are not explored in this research, such as the appearance of new targets, or changes in overall situation like weather.

3.3 Summary

Strike Ops is a simple real-time strategy game with strong elements of resource allocation, spatial reasoning, and anticipation. Blue plays with complex dynamics and
compromises between optimally allocating the assets provided and producing routes that minimize exposure to risk. The element of trapping provides a challenging aspect, as both players attempt to out anticipate each other. Strike Ops was also designed to have very little micromanagement, so that players win only through long term strategy. These complications make the game interesting, the underlying resource allocation problems difficult, and thus suitable for genetic and evolutionary approaches.
Chapter 4
GAP - The Genetic Algorithm Player

We play strategy games by casting them as optimization problems, which we then use genetic algorithms to solve. Genetic algorithms require only an encoding and a fitness function to function effectively, both of which can be produced without extensive expert knowledge about how to play the game. We develop GAP, the Blue Genetic Algorithm Player, to play the attacking player (blue) in Strike Ops. GAP works by applying its GA to the given scenario as shown in Figure 4.1. The GA creates populations of bit strings, which are converted into plans and evaluated. Based on this evaluated fitness, individuals are recombined and new plans are produced. We combine a steady state population model, roulette wheel selection, two point crossover and bitwise mutation to form our GA. When the population converges, it produces a good allocation with corresponding routes which can then be used to effectively play the game. Results show that the plans produced are near optimal with respect to the knowledge known to the GA at that time.

GAP encodes potential plans of actions as bitstrings, searching towards the bitstring containing the best plan of attack. Each bitstring should encode solutions for all of the strategic decisions required to play the game. GAP considers the answers to the following decisions as it plays the game.
Figure 4.1  GAP’s Search

- Resource allocation
  - Which targets to attack?
  - Which platforms to attack them with?
  - Which weapons to use on them?

- Spatial Reasoning
  - How to route platforms to targets and back?

- Anticipation
  - Where are there likely to be traps?
  - How should platforms be routed to avoid them?
4.1 Encoding

GAP encodes two pieces of data which are used to make the necessary game-playing decisions.

1. Which assets to use on which targets.

2. How to route platforms in order to carry out the allocation.

The allocation of weapons to targets explicitly answers the question of which weapons to use on which targets, while implicitly determining both which targets to attack and which platforms to attack them with (each platform carries a fixed set of assets). With information about how to route the platforms, and knowledge about which platforms must reach which targets we can resolve the issue of routing platforms to the targets and back. If our routing algorithm is clever it will anticipate opponent traps, answering the questions about anticipation - covered in detail in chapter 6.

These two fundamental pieces of information both represent optimization problems, with non-linear interdependencies between them - a good allocation might require a very risky route, and a very safe route might not come within range of valuable targets. By encoding and searching for solutions to these questions in parallel as shown in Figure 4.2, we can resolve their inherent interdependence - searching towards near-optimal allocations and their corresponding routes.
The resource allocation can be encoded by reducing it to an enumeration of assets to targets and encoding those in a bitstring. The top of the allocation section illustrates the allocation of asset A1 on platform P1 to target T4, asset A2 to target T3 and so on. Tabulating the asset to target allocation gives the table below. The routing information GAP needs to produce is a set of waypoints for each platform to follow so that it goes from its starting point to each target and back while minimizing risk by avoiding threats. We could encode the waypoints into the bitstring, but that would greatly increase the size of our search space. A* was chosen as it has been shown to always produce optimal routes, and has been used very widely in real-time strategy games. We use a parameterized form of A* to produce the routes for each
aircraft, and we encode the parameters to the pathfinder into the bitstring. Each encoding then has an exact allocation of assets to targets, combined with guidelines for how it wants aircraft to be routed to fulfill the allocation.

For routing information the GA encodes a single parameter, RC, which is described in the section 4.3. The A* router below uses this parameter along with the allocation in order to produce routes. We encode RC by using a standard fixed precision binary encoding. RC is encoded as a binary fraction between empirically chosen min/max values.

4.2 Routing

![Diagram of routing process](image.png)

Figure 4.3  How Routes are Built From an Encoding.

GAP must be able to produce routing data for Blue’s platforms in order to play the game. Figure 4.3 shows how the A* algorithm is used to build routes [29]. From the allocation of assets to targets we produce an ordered list of waypoints for each platform to visit. In order for platform P1 to use asset A1 on target T4, it has to fly to
T4’s location. Applying this to all of P1’s assets produces a list of waypoints for P1 to visit during its mission. From the list of waypoints we use a path-finding algorithm to produce more intelligent routes. A number of path-finding algorithms exist, A* was chosen as it is very widely used and comprises the majority of path-finding algorithms in games.

In order to find a route from some start point to a goal point we first convert the continuous world into a graph. Each node in the graph represents a position in the world, and the edges have weights representing the costs associated with moving between those points. In the earlier phase of the work the graph was built by discretizing the world into voxels. This technique had several complications, mainly that it required a post smoothing phase to overly avoid orthogonal movement. Later we produced the graph by creating nodes at points of interest - the start/goal point and around the radii of threats. All nodes have edges between them, and the costs for those edges is computed based upon the risk presented between them. The graph produced is visualized in Figure 4.4 Any optimal path with either be directly from start to goal, or will skirt the outskirts of threats along the way, so this produces optimal routes much faster and without the need for smoothing as with voxels.

Edge costs in the graph are computed based upon the risk involved in flying between the nodes, nodes that fly through threats have much higher risk then those that do not. Once the graph has been produced A* works by taking the starting node
Figure 4.4  A Pathfinding Graph
and calculating costs to each neighboring node. Comparing the cost of getting to that
node from the start location (risk it presents + distance to it) with an estimate of
its distance to the goal, A* produces a value for each node. Those possible neighbor
nodes put into a "open" list sorted according this value. The process then repeats on
the most promising node until the destination is reached. A* is guaranteed to always
find the shortest route if it is given a proper underestimate of distance to the goal,
which we have.

4.3 General Routing Knowledge

A* is shown to always find the optimal route based on its cost functions, in our
case the shortest route avoiding known threats. Since our game includes traps, which
are unknown at the time of routing, the shortest route is not always desirable. In order
to do more interesting routing, we must be able to bias A* towards producing routes
that are longer, or more dangerous then those immediately apparent. We do this by
modifying the graph A* searches, which results in a variety of effects on the kinds of
routes. For example penalizing each node based on how far south it is provides a bias
that tends to produce north traveling routes, thus producing an overall strategy of
attacking from the north. Routing two groups of platforms, one with a southern bias
and one with a northern bias is likely to produce pincer attacks. However our goal
is to avoid traps, and we note that traps are most effective in confined areas. The
human in our game is also trying to avoid confined areas, and to do this we need to
modify the nodes in order to identify areas that are confined. This notion that we might want to avoid confined areas is the only game specific knowledge we are using in our implementation of GAP. This results from the fact that the encoding needs to be able to contain possibly important strategic information. It is, however, significantly easier to determine which kinds of strategic notions might be useful then it is to gain enough expert knowledge to know how to use each idea - we do not need to experiment and determine the values of this parameter, or how it relates to other parameters as we would in a more traditional system, we only need know that this parameter may be useful. In our representation we identify these confined areas by extending the effective radii of threats when we build the graph. The extension is calculated by a simple multiplication of each radius by a coefficient $RC$, which determines the kind of routes produced. Figure 4.5 shows the effect $RC$ has on routing. When $RC < 1.0$: the radii of the threats shrink, and the routes produced tend to be very direct - coming inside the boundaries of threats to save time. When $RC = 1.0$: the radii stay the same, and the routes produced skirt the outsides of the threats. When $RC > 1.0$: the radii expand and overlap, and the routes produced avoid previously confined areas - taking long circuitous routes to avoid risk. As encoded, $RC$ uses 8 bits to produce a range from 0 to 3, which was empirically chosen.
4.4 Fitness

With the encoding in place we can now generate populations of individual plans. In order to search towards good plans we need an evaluation function. We evaluate the fitness of an individual in GAP’s population by running the game and checking the outcome. Blue’s goals are to maximize damage done to red targets, while minimizing damage done to its platforms. Shorter simpler routes are also desirable, so we include a penalty in the fitness function based on the total distance traveled. This gives the fitness calculated as shown in Equation 4.1

\[ fit(\text{plan}) = \text{Damage(Red)} - \text{Damage(Blue)} - d \times c \]  

(4.1)

\( d \) is the total distance traveled by Blue’s platforms and \( c \) is a coefficient to scale the penalty appropriately. Total damage done is calculated below.

\[ \text{Damage(\text{Player})} = \sum_{E \in F} E_v \times (1 - E_s) \]
$E$ is an entity in the game and $F$ is the set of all forces belonging to that side. $E_v$ is the value of $E$, while $E_s$ is the probability of survival for entity $E$. We use probabilistic health metrics to evaluate entity damage.

### 4.4.1 Probabilistic Health Metrics

In many games, entities possess hit-points which represent their ability to take damage. Each attack then removes a number of hit-points and when reduced to zero (0) hit-points that entity is destroyed. In reality, weapons have a more hit or miss effect, whereby they entirely destroy things or leave them functional. A single attack may destroy an entity or multiple attacks may have no effect. This paradigm introduces a high amount of stochastic error into the game. Evaluating a plan can result in outcomes ranging from total failure to perfect success, which makes it difficult to compare two plans. By taking a statistical analysis we achieve better results. Consider the state of each entity at the end of the mission as a random variable. Identifying the expected values for those variables becomes one means to judge the effectiveness of a plan. Ideally we would like to know that if we carry out plan A we have a 65 chance of destroying the target, while with plan B we have an 85 chance. These expected values can be estimated by playing a number of games for each plan and averaging the results. However, doing multiple runs to determine a single evaluation increases the computational expense many-fold.

We use a different approach based on probabilistic health metrics. Instead of
monitoring whether or not an object has been destroyed, we monitor the probability of its survival up until that point in time. Being attacked no longer destroys objects and removes them from the game, it reduces their probability of survival from then on according to Equation 4.2.

\[ S(E) = S_{t_0}(E) \times (1 - D(E)) \]  

(4.2)

\( E \) is the entity being considered, which is a platform or target under attack. \( S(E) \) represents the chance of that entity surviving past this point in time. \( S_{t_0}(E) \) is chance of survival up until the attack. \( D(E) \) is the chance of that platform being destroyed by the attack as given by equation 4.3.

\[ D(E) = S(A) \times E(W) \]  

(4.3)

\( D(E) \) is the chance of destruction by this attack. \( S(A) \) is the attackers chance of survival up until the time of the attack. \( E(W) \) is the effectiveness of the attackers weapon as given by the weapon-target effectiveness table. This method gives us the expected values of survival for all entities in the game within one run of the game, thereby producing a representative and non-stochastic evaluation of the value of a plan. As a side effect, we also gain a smoother gradient for the GA to search as well as consistently reproducible evaluations.
4.5 Results - GAP Can Play the Game

With both an encoding and a fitness function GAP can use its genetic algorithm to search towards optimal plans of attack. To test this ability we setup a mission for GAP to play, and graph the fitness of individuals within its population. The mission chosen is shown in figure 4.6

![Figure 4.6 The Mission](image)

This mission was chosen to be simple and to have easily analyzable results. The
mission takes place in Northern Nevada and California, with Walker lake visible near the bottom of the map. Blue possesses one platform which is armed with 8 assets (weapons) and the platform takes off from and returns to the lower left hand corner of the map. Red possesses eight targets distributed in the top right region of the map, and six threats that defend them. The first stage in Blue’s planning is determining the allocation of the eight assets. Each asset can be allocated to any of the eight targets, giving $8^8 = 2^{24}$ allocations. GAP plays the mission 50 times, and we graph the average fitness of individuals inside the population against their generation in Figure 4.7. The graph shows a strong approach toward the optimum, which was brute force located at 252. GAP approaches within 5% of optimal allocation and routing 95% of the time. This indicates that GAP can form effective strategies for playing the game.
Figure 4.7  Best/Worst/Average Individual Fitness as a function of Generation - Aver-aged over 50 runs.
Chapter 5
Dynamism and Re-planning

We have shown that GAP can take the initial scenario and produce near-optimal plans of actions. These plans are near-optimal with respect to the knowledge available at that time, they are often non-optimal when unexpected changes occur. This results from the fact that we plan without perfect knowledge of the game-state and how that game-state will change in the future as a result of opponent moves. To evaluate potential plans of action we evaluate them against assumed opponent moves, directing the search towards good counter strategies for those opponents. Opponents which deviate from our assumptions are no longer being optimally planned for; leading GAP to be baited, trapped, and otherwise deceived. Strike Ops is a dynamic imperfect knowledge game, and to deal with these changing game-states we first utilize re-planning. During the game, whenever GAP encounters an important and unexpected change in the game state, such as a trap appearing, it redoes its planning taking the new situation into account. We take the game as a series of game-states, where an unexpected change produces a new game-state as shown in Figure 5.1. At each game-state GAP runs the GA to produce a plan of action which is carried out until the next game-state is reached.

Unexpected opponent moves or changes in game-state leads GAP to rerun the
Figure 5.1 Considering the continuous game as a discrete series of game states

GA, which produces a new plan in response to the new situation. The GA produces near-optimal solutions as before: avoiding pop-up threats, disengaging from targets have become too well defended, or re-prioritizing towards new high-value targets.

Figure 5.2 Yellow path was re-planned when the pop-up occurred

With re-planning GAP plays an effective reactive game, at each step determining near-optimal courses of actions with respect to currently available knowledge. While the re-planning responds effectively to changes in situation, it suffers from some lim-
itations. The primary problem is that genetic algorithms are slow, and GAP takes significant computation to respond to changes in situation. Strike Ops is a real-time strategy game so while GAP is re-planning the game continues to play. The game-state could change significantly, to the point of game being lost, while GAP waits on its re-planning. To overcome this limitation we use case-injection to improve the search time of GAP’s GA. This work was published in [30].

Case-injected GA’s have been shown to increase performance at similar problems as they gain experience. In strategic games such situations occur often, as every opponent action and situational change is a change in game state. The new game-states are usually similar to the previous game state, a few new targets or threats, maybe a change in weather. These changes are often minor as few opponent actions are worth the effort of redeveloping your entire strategy from scratch. Usually adapting your previous strategy to the new situation is faster, more consistent, and more effective. Case injection maintains information from previous strategies, allowing us to keep our previous strategy in mind when developing new ones. Case injection thus allows the GA to adapt old strategies to new situations, maintaining past knowledge that is still applicable, while redeveloping aspects of the old strategy to cope with unforeseen situations.
5.1 Case Injection for Re-planning

We use case-injection to improve re-planning speed. As the GA plans a mission it extracts good individuals and saves them to a case-base. Then, if a new target or threat appears, the GA re-plans to produce a new plan that responds to those changes. This time however it injects material from the case-base, maintaining some of the information gained from the previous searches. The effect is that of reconsidering what was previously successful. With each change in game-state change we re-plan, continuing to extract more material to the case-base. As the game continues, the GA responds faster and better, remembering a range of possibilities that were previously effective. Figure 5.3 combines Figure 5.1 and Figure 2.3 to show how case-injection is used for re-planning.

We use standard case-injection, replacing the worst 10% of the population every log(numgens) generations. The cases we inject are probabilistically chosen based upon hamming distance to the current best, and extraction saves every individual that improves on the previous best in population.

5.2 Results

We test the effectiveness of case-injection for re-planning under two dynamics. We first explored case-injection’s ability to scale from missions involving a few aircraft and targets, up to missions involving large numbers of platforms, assets, targets and threats. We explore this in section 5.2.1. Secondly, we explored how the scope of
the change between game-states effects case-injections effectiveness. If the opponent makes a move requiring the total reconstruction of the attack plan then case-injection should be less effective than if the previous plan needs only to be slightly adapted. We explore this in section 5.2.2.

5.2.1 Game Complexity

Game complexity refers to the complexity of the individual mission being played. Increasing the resources available for each side to allocate increases the strategic search space presented to each player. How does this increase in search space alter the effect of case injection on the genetic search?

To test this, we first constructed 3 missions of increasing complexity. More complex missions have more attacking aircraft loaded with more weapons to attack more targets. The mission’s general character does not change. The defending player follows a script activating pop-up threats early in the mission leading the attacking
player (GAP) to re-plan its attacking strategy. Scripting keeps the analysis straightforward and repeatable. We then let the GA play this game multiple times, with and without case injection and analyze the GA’s response. We provide mission profiles below.

- **Simple Mission**
  - 4 platforms, 8 assets, 8 targets
  - 30 bits per chromosome

- **Medium Mission**
  - 6 platforms, 12 assets, 20 targets
  - 66 bits per chromosome

- **Large Mission**
  - 10 platforms, 20 assets, 40 targets
  - 114 bits per chromosome

We run a GA with and without case injection on each of the three missions fifty times with different random seeds and plot the average number of evaluations made in Figure 5.4-left. Note that **Case-Injected Genetic Algorithm (CIGAR)** reduces the number of evaluations required to converge, out-performing the non-injected GA.
As the missions become more complicated case-injection provides increasingly larger gains over the non-injected GA. On more complicated missions CIGAR retains more information, giving it a larger advantage over the GA.

We can also see that although CIGAR takes less time to converge, the quality of solutions produced by CIGAR does not suffer. Figure 5.4-right plots the average of the maximum fitness found by the GA or CIGAR over fifty runs. Both CIGAR and the non-injected GA search until they find near-optimal solutions, but case-
injection provides a significant reduction in the computation required to reach those near-optimums.

5.2.2 Re-planning Scope

GAP re-plans whenever the situation changes. These changes range from minor events like discovering a poorly valued target to big events like highly time-critical and important targets appearing. Case-injection exploits information gained in previous searches, and the scope of the situation change determines how much of that previous knowledge is pertinent to the current situation. How does case-injection work under these different kinds of changes?

We again construct three missions, this time with different defending layouts and scripted actions for the defending player. In each mission the defending player makes five changes, these changes having an increasing impact on the attacking players strategy. We summarize the missions and GAP’s response below.

- Simple Re-plan, shown in Figure 5.5 - Weak short-range threats pop-up on the way to targets (same as previous 3 missions).
  
  - GA reroutes to avoid new threats
  
  - Minor changes to weapon-target allocation

- Moderate Re-plan, shown in Figure 5.6 - Medium-range threats pop-up around a handful of targets
Figure 5.5  Simple mission re-planning scenario

Figure 5.6  Moderate mission re-plan scenario
- Moderate changes in allocation
- Avoids newly protected low value threats
- Redirects additional attackers to newly protected high value targets
- Significant routing changes to avoid new threats / reach new targets

- Complex Re-plan, shown in Figure 5.7 - Powerful large-range popups occur defending a large cluster of targets

![Figure 5.7](image)

Figure 5.7  Complex mission re-plan scenario

- Large changes of allocations
- Wings (groups) of aircraft diverted from new hot zones
- Focusing of aircraft towards the most highly valued targets
- Rerouting of most aircraft for each re-plan
Figure 5.8-left shows the number of evaluations required as a function of the number of re-plans for the above missions. As the scope of the re-plan increases, case-injection’s advantage decreases. In other words, case-injection focuses search towards previously successful solutions. As the new solution moves further from the old solution the advantage provided by case-injection decreases. Figure 5.8-Right shows once again that CIGAR’s speed advantage does not come at the expense of lower quality solutions. On the contrary the figure shows that CIGAR produces better quality plans. CIGAR’s speedup over the GA is statistically significant. The fitness gains through case injection, while consistent, are comparably small and difficult to show as statistically significant without a large number of runs.

We have shown that a genetic algorithm can play computer strategy games by solving the sequence of underlying resource allocation problems. Case-injection causes a statistically significant improvement in the speed with which our genetic algorithm player can respond to opponent actions and other changes, without negatively effecting the fitness of solutions produced. We explored the effects of mission complexity and re-planning scope, showing that the advantage provided by case injection increases as the mission becomes more complicated, and decreases as the difference between new and old situations grows. Note that case injection still provides a significant improvement even when the game situation changes drastically. Playing RTS games with a GA presents a good application of case injection, and we have explored
Figure 5.8  Re-planning Size versus Evaluations Required - Left and Fitness - Right
how case-injection impacts the dynamics of the game, showing significant improvement in response time.
The third set of fundamental decisions are those regarding anticipation. What kind of units is my opponent building? Which of my units should I use to counter them? We explore this idea of anticipation by looking at traps. Traps are difficult and interesting to deal with because they are strongly rooted in anticipation. Where should I lay traps? Where has my enemy put there traps? Both laying and avoiding traps requires anticipation, figuring out both where your opponent will be, and where your opponent expects you to be. A complex and difficult problem quickly emerges with no easy solution. Our goal is for GAP to learn from experience, both from its own and others to avoid traps.

We considered two possibilities for adapting GAP to deal with traps. The first being to construct a model of our opponent, which could be used to anticipate what kind of moves our opponent would be likely to make. Then we could utilize this information by including anticipated opponent moves in our fitness function. Individuals would then receive higher fitness if they played in anticipation of past opponent moves. The GA would then search towards plans that were effective and anticipatory. For example remembering that our opponent is weak to his left, and including that weakness in our search function that so future searches will prefer plans which attack from
the left. This method requires a system which models the opponent, determining which moves they are likely to make based upon the game-state. The development of an opponent modeler would require significant additional knowledge about how to play the game as our opponent. The second option was to remember strategies of ours which anticipated opponent moves. An example is just to remember to attack from the left, and to prefer those plans in the future. This avoids the need to model our opponent, requiring only a way to store information on what kind of plans we want, methods to acquire that knowledge, and the means to apply the knowledge. While the first technique seemed more natural, the second is significantly simpler and requires less expert knowledge. Because of these expected advantages we chose to implement the second technique, directly storing and applying ”what we should have done” knowledge.

GAP maintains a database of effective past plans containing important anticipatory knowledge we would like to include in future plans. By biasing GAP to use that information we make future searches produce plans that play in a more anticipatory way. The application of this knowledge is done by using case-injection in a novel way.

Case-injection is generally used to improve performance, as an example our work in Section 5.1 to improve re-planning speed. Case-injections has a side-effect of biasing search towards injected material. By injecting material containing the kind of strategies we want - anticipatory past plans, we make that injected material more
likely to be expressed in the final plan. A plan with important information we would like to learn from (like how to avoid a trap) takes the form of a case in the case-base. To apply that knowledge we inject it into the population, where it biases the search to likely contain that information in the final solution. Case injection provides an implementation of these steps: building a case-base of individuals from past games stores important knowledge, the injection of those individuals applies the knowledge towards future search. The cases can come from anywhere, so long as they contain useful information.

6.1 Reflecting on Past Games

The first source of cases is from past play. As GAP plays against opponents, it can learn over the long term by building a case-base containing anticipatory information from past games. GAP records games played and runs offline in order to determine the optimal way to have played past games. These ”how we should have played” games are stored in the case-base, where they can be injected in the future. In order to determine how we should have played we replay the game, but we take the opponent moves into account when calculating fitness as shown in Figure 6.1. The simulation now contains knowledge about opponents moves, in our case, the game pop-up traps. We do not include the traps as part of the information given to GAP when it produces the plans however, it is only included in the evaluation. This means that individuals who take the original game state and produce plans which anticipate those popups
will receive the highest fitness. From this the search will progress towards the best anticipatory plans, and we can extract individuals to the case-base along the way. When faced with other opponents, GAP then injects individuals from the case-base, biasing the current search towards containing this learned anticipatory knowledge.

![Figure 6.1 Reflection Architecture](image)

**Figure 6.1** Reflection Architecture

### 6.2 The Scenario

To test GAP ability to learn from experience we play the mission shown in Figure 6.2. The platforms start from the lower left, the targets are on the left hand side. There are two good routing options for reaching the targets, a direct route (YELLOW) through the confined corridor and a circuitous route (GREEN) which goes the long way around. The third option, a BLACK route flies through known targets because it has a low RC, and gets low fitness. GAP plays the scenario, where it usually produces yellow routes - falling into traps red has laid in the center. GAP then
replays the game, determining it should have played green - extracting and storing good green plans into the case-base as it searches. Saving individuals to the case-base from this search stores a cross-section of plans containing "trap avoiding" knowledge.

![Figure 6.2](image)

**Figure 6.2** Left - The Trapping Scenario, Right - Routes

The process produces a case-base of individuals containing important knowledge about how we should play, but how can we use that knowledge in order to play smarter in the future? Case Injection has been shown [15] to increase the search speed and the quality of the final solution produced by a GA working on a similar problem. It also tends to produce answers similar to old ones by biasing the search to look in areas that were previously successful – exploiting this effect gives our GA its learning behavior. When playing the game we periodically inject a number of individuals from the case-base into the population, biasing our current search towards information from
those individuals. Injection occurs by replacing the worst members of the population with individuals chosen from the case database through a "Probabilistic Closest to the Best" strategy [11]. Those individuals bring their "trap avoiding" knowledge into the population, increasing the likelihood of that knowledge being used in the final solution and therefore increasing GAP’s ability to avoid the trap.

6.3 Results - Learning From Experience

We present results showing that with case injection GAP learns to avoid the trap. We also analyze the effect of altering the population size and number of generations on the strength of the biasing provided by case injection.

GAP’s ability to learn to avoid the trap is shown in Figure 6.3. The figure compares the histograms of $RC$ values produced by GAP with and without case injection. Case injection leads to a strong shift in the kinds of $RC$’s produced, biasing the population towards using green routes. The effect of this bias being a large and statistically significant increase in the frequency at which strategies containing green routes were produced ($2\% \implies 42\%$). These results were based on 50 independent runs of the system and show that case injection does bias the search toward avoiding the trap.

Figure 6.6.2-left compares the fitness’s with and without case injection. Without case injection the search shows a strong approach toward the optimal yellow plan; with injection the population quickly converges toward the optimal green plan. Case injection applies a bias towards green routes, however the GA has a tendency to act
in opposition of this bias, trying to search towards ever shorter routes. The ability of
the GA to overcome the bias through manipulation of injected material is dependent
on the size of the population and the number of generations it runs. Figure 6.4-Right
illustrates this effect. As the number of evaluations allotted to the GA is increased, the
frequency of green routes being produced as a final solution decrease. Counteracting
this tendency requires a careful balance of GA and case-injection parameters.

![Figure 6.3 Histogram of Routing Parameters produced without Case Injection.](image)

6.4 Learning from Others

By re-planning past games we can extract cases containing information about how
we should have played before. Injecting that information biases future gameplaying to
play more like these "should haves". As the extracting and elicitation of knowledge
is separate from the application, we can apply knowledge gained from any source
with similar effect. Instead of learning from our own experience we can learn from
others. Imagine playing a game and seeing your opponents do something you had
Figure 6.4  Left: Effect of Case Injection on Fitness Inside the GA over time Right: Effect of Population Size and the Number of Generations on Percentage Green routes Produced

not considered that worked out to great effect. Seeing something new, you are likely to try to learn some of the dynamics of that move so you can incorporate it into your own play and become a more versatile player. Ideally you would like perfect understanding of when and where this move is effective and ineffective, and how to best execute the move under various circumstances. Whether the move is using a combination of chess pieces in a particular way, bluffing in poker, or doing a reaver drop in Starcraft the general idea remains. In order to imitate this process we use a two step approach with case injection. First we learn knowledge from human players by saving their decision making during game play and encoding it for storage in the case-base. We apply this knowledge by periodically injecting these stored cases into GAP’s evolving population as we did when learning from experience.
We would like to learn from other players. To do this we observer them play and record every move they make. However in order to apply knowledge it needs to be in the form of a case. If our encoding was a direct encoding of moves this would be relatively straightforward, but our encoding encapsulates a general idea - RC. In order to convert the human strategy into a chromosome we use a reverse engineering technique.

6.4.1 Reverse Engineering

The goal of reverse engineering is to convert the game play from the other player into a chromosome which contains all of its important strategy elements. This is a non-trivial task. To do this we run GAP, only we use a similarity metric to the given plan as a fitness function. GAP will then evolve towards the most similar plan. Our similarity metric based upon a direct comparison of the allocation and a distance measures between platforms following the two routes. The higher fitness goes to plans which have a more similar allocation, or which route aircraft more similarly to the human plan. The result of this is the transformation of of the given human route, the white line in Figure 6.5, into a chromosome representing the plan in the green line in Figure 6.5. The plans are not identical because the chromosome does not contain exact routing information, it can only approximate it by adjusting RC. Note the overall fitness difference between these two plans is less then 2%. This can then be extracted to the case-base, where it can be injected in the future.
6.5 Results - Learning From Others

We show results that the GA can play the game, and by using case injection we can significantly increase its likelihood of playing like a human. The mission being played is shown in Figure 6.5 - Left. This mission was chosen to be simple, to have easily analyzable results, and to allow the GA to learn external knowledge from the human. As many games show similar dynamics, this mission is a good arena for examining the general effectiveness of using case injection for learning from humans. The mission is the same one used in section 4.5, it takes place in Northern Nevada and California, with Walker Lake visible near the bottom of the map. In our work, knowledge acquisition takes the form of building a case-base of chromosomes representing past strategies used by human experts. Each strategy should be represented in a general way, so that it can be applied robustly across a variety of missions. RC allows us to represent the knowledge of avoiding confined areas as defined by the expert in our mission.

6.6 The Mission

Blue possesses one platform which is armed with 8 assets (weapons), the platform takes off from and returns to the lower left hand corner of the map. Red possesses eight targets distributed in the top right region of the map, and six threats that defend them. The first stage in Blue’s planning is determining the allocation of the eight assets. Each asset can be allocated to any of the eight targets, giving $8^8 = 2^{24}$
allocations. The second stage in Blue’s planning involves finding routes for each of the platforms to follow during their mission. These routes should be short and simple but still minimize exposure to risk. We categorize Blue’s possible routes into two categories. Yellow routes fly through the corridor between the threats, while green routes fly around. The evaluator has no direct knowledge of potential danger presented to platforms inside the corridor area. Because of this, the evaluator optimal solution is the yellow route, since it is the shortest. The human expert however, understands the potential for danger as the corridor provides the greatest potential for a pop-up trap. Knowing this a green route is the human optimal solution - the plan produced by the human is shown as the white line in Figure 6.5 - Right. The human plan was observed, and then reverse engineered into a chromosome, which was stored in the case-bas. We then ran GAP on this mission, injecting from the
case-base and observing the results.

The category of routes produced is determined by the values of $RC$. GAP’s ability to produce the human-like route (green) is based on the values of $RC$ it chooses. Figures 6.6 and 6.7 show the distribution of $RC$ produced by the non-injected genetic algorithm and the case-injected genetic algorithm. Comparing Figure 6.6 with Figure 6.7 shows a significant shift in the $RC$’s produced, which leads to a large increase in the number of green routes generated by the case injected GA. Without case injection GAP produced no green routes, using case injection biased GAP to produce 64% green routes, this difference is statistically significant. These results were based on 50 different runs of the system with different random seeds and show that case injection does bias the search towards the human strategy.

![Histogram of Routing Parameters produced without Case Injection.](image)

**Figure 6.6** Histogram of Routing Parameters produced without Case Injection.

### 6.6.1 Alternative Mission

Moving to the mission shown in Figure 6.8 and repeating the process produces the histograms shown in Figures 6.9 and 6.10. The same effect on $RC$ can be observed
even though the missions are significantly different, and even though we use the human plan from the previous mission. Our general routing representation allows GAP to learn to the lesson of avoiding confined areas from the human expert.

6.6.2 Bias Strength

Case injection applies a bias to the GA search, the number and frequency of individuals injected determines the strength of this bias. However the fitness function also contains a term that biases against producing longer routes. As the number of evaluations allotted to the GA is increased, the bias against longer routes outweighs the bias towards human strategies and fewer green routes are produced. The effect is shown in Figure 6.6.2.

6.7 Fitness Inflation

In both learning from experience and learning from others we noticed that injected material did not always find its way to the ultimate solution. The GA has a tendency to take injected material and optimize away the important lessons in order to get a
Figure 6.8  Alternate Mission

Figure 6.9  Histogram of Routing Parameters produced without Case Injection on the Alternate Mission.
Figure 6.10  Histogram of Routing Parameters produced with Case Injection on the Alternate Mission.

Figure 6.11  Number of Evaluations effect on the Percentage Green routes Produced
higher fitness. By reducing the number of evaluations we found we could limit this, so that the GA just had time to tune the most important changes to the chromosome. But this required very precise settings of parameters and was relatively unstable. In order to present a stable situation we introduced the concept of fitness inflation. In this we track injected material in the population, and inflate the fitness of individuals containing it. The amount of inflation is determined by a coefficient, with small values of this the GA will make any change to the chromosome that gives a moderate improvement in fitness, with large values the GA will make only the changes which drastically improve the fitness. By doing this we can maintain injected material in the population, which greatly improves the performance of GAP at avoiding traps.

To show this we ran GAP on the alternative mission 6.6.1, injecting the case taken from the human player which biased it towards green routes. Our fitness inflation coefficient is at 20%, so a completely injected individual has a 20% advantage of a non-injected one. We graph the densities of RC’s produced over 50 runs with and without fitness inflation in Figure 6.12.

Without case-injection about 80% of the runs fall into the trap. With case-injection and a case-base with trap-avoiding individuals the performance improves with about 60% avoiding the trap. However a large number of runs still end up in the trap, because the GA tunes the RC into a yellow route for that extra 1% fitness. With fitness inflation however that information is maintained, as GAP almost always
avoids the trap. GAP is still changing the majority of the injected individual, the only information being maintained consistently is the injected material which has the least effect on the fitness, namely $RC$. Fitness inflation allows GAP to maintain injected information, helping it resist the tendency to search towards higher fitness in order to preserve important information not contained in the fitness function. With it GAP can consistently learn to avoid traps, and can effectively learn from other players.
Chapter 7
Conclusions

By casting the game as a resource-allocation problem which is searched with a genetic algorithm, GAP is capable of effectively playing our real-time strategy game. Results show quick convergence to near-optimal plans of attack, combining effective resource allocations with good coordinates routes. The dynamic nature of the game can be addressed with a reactive re-planning system, which replans to produce near-optimal responses to changes in situation. Results show case-injection can be used to greatly improve the performance of this system, significantly reducing the computation time required. Case-injection also provides answers for the difficult question of anticipation. By using reflection and reverse engineering the system can learn both from its own past experience, and from the experiences of others. This leads to effective anticipatory play. Results show improvement with each game played, leading to more effective plans that avoid enemy traps. Results also show that GAP can be biased to play like a human player, absorbing important aspects of the human’s strategy into GAP’s strategy.

The results indicate that GAP competently answers of all the fundamental game-playing decisions, forming an effective game-player for Strike Ops. GAP’s architecture requires little expert knowledge to function, so our approach should generalize well to
other applications. We expect many of our techniques to be broadly applicable within the domains of both RTS games, and their corresponding real world applications.

This research has several avenues for interesting future work. By shifting to a more dynamic game involving resource gathering and unit construction we allow for the development of longer term strategies. Shifting away from the evolution of individual plans towards the evolution of complete game strategies allows a number of benefits. Firstly a single strategy can play a complete dynamic game without having to be replanned, freeing us from that significant computational burden. Secondly since strategies can now evolve over the course of many games we can show true long term evolution. Currently the system evolves for the current game, but takes into account some information from past games.

To deal with the more dynamic game-world we have shifted away from searching for individual game plans and towards the evolution of complete game-playing strategies. Encoding a complete strategy into a bitstring is a daunting task, that is being approached by encoding influence map trees. Each individual in the population can then be used to play an entire game, without having to rerun the genetic algorithm during the game. This also allows long term evolution, in the sense that strategies can be evolved over games instead of having really a singular strategy that can draw from elements learned from past play as was done with Strike-Ops. By encoding strategies we can then shift to co-evolution, whereby we play individuals against one another.
The goal of co-evolution is an increasing spiral of confidence, where players continue to improve and evolve against one another. This would bring us close to our ultimate goal, the ability to create and evolve game-players.
References


