Simulation-based Optimization of StarCraft Tactical AI through Evolutionary Computation

Nasri Othman, James Decraene, Wentong Cai, Member, IEEE, Nan Hu, Malcolm Yoke Hean Low, Senior Member, IEEE, and Alexandre Gouaillard

Abstract—The development of competent AI for real-time strategy games such as StarCraft is made difficult by the myriad of strategic and tactical reasonings which must be performed concurrently. A significant portion of StarCraft gameplay is in managing tactical conflict with opposing forces. We present a modular framework for simulating AI vs. AI conflicts through an XML specification, whereby the behavioural and tactical components for each force can be varied. Evolutionary computation can be employed on aspects of the scenario to yield superior solutions. Through evolution, a StarCraft AI tournament bot achieved a success rate of 68% against its original version. We also demonstrate the use of evolutionary computation to yield a tactical attack path to maximise enemy casualties. We believe that our framework can be used to perform automatic refinement on AI bots in StarCraft.

I. INTRODUCTION

Development of AI agents for real-time strategy games which can competently rival human players remains a difficult problem [1]. A StarCraft game is highly complicated and is determined by a multitude of factors, including the properties of the game units, the map used and the randomised starting positions of the players. An AI agent must simultaneously balance the tasks of scouting, resource gathering, base building, troop production and tactical combat to achieve victory over an adversary.

The performance of an AI agent can be improved by tuning numerous parameters. But, how can we find the optimal parameters to make it decisively stronger? An AI developer who manually tunes the parameters of an agent would have to anticipate all possible game states and situations [2]. Instead, metaheuristics and machine learning techniques can be used to iteratively improve the agent behaviour against a measure of fitness.

Evolutionary computation (EC) is a popular metaheuristics approach which has been demonstrated to rapidly generate good solutions in numerous domains [3]. Increasingly, it is used to tune AI agents in strategy games such as Chess [4] and Wargus [5]. To apply evolutionary computation in StarCraft, an agent implementation must be integrated with an external EC framework, which can automatically explore the decision space of evolvable parameters, evaluate candidate solutions under a given scenario, and return the solutions which yield the best performance under that scenario. Furthermore, it is pertinent to minimize the evaluation time of a scenario so that the optimization process can complete in reasonable time as a full game of StarCraft (i.e., comprising the resource gathering, production and warfare stages) may typically take over 20 minutes to complete.

In this paper, we present a modular framework to rapidly simulate StarCraft AI vs. AI tactical scenarios through an XML specification whereby aspects such as the behaviour and tactical composition of each side can be varied. These tactical scenarios comprise only of two opposing forces with fixed configurations (i.e., fixed armies and positions) and do not involve other aspects of StarCraft such as resource management and production (i.e., a comprehensive StarCraft game). A speed-up technique applicable to StarCraft is employed to significantly reduce the evaluation time of a scenario. By limiting the scope of the simulation and reducing its evaluation time, automatic refinement of StarCraft tactical AI through evolutionary computation can then be achieved.

Through our case studies, we demonstrate that optimization of StarCraft AI is feasible through evolutionary computation and can be achieved in reasonable time. In the first study, we evolved the real-valued behavioural parameters of FreScBot, the champion bot of Tournaments 1 & 2 of the AIIDE 2010 StarCraft AI Competition, and have improved its success rate to 68% against its original version. In the second study, evolutionary computation was used to yield an attack path which maximises enemy casualties in a test scenario. While our approach is limited in scope and only considers a limited set of StarCraft gameplay, it can demonstrably be used to improve the performance of bots in tactical combat.

This paper is organised as follows: Section II presents StarCraft and related work. In Section III, we introduce our framework and its architecture. Section IV details how to evolve scenarios in our framework using evolutionary computation. Section V illustrates two case studies demonstrating evolutionary computation in StarCraft. Lastly, we discuss future work and conclude this paper.

II. RELATED WORK

This section discusses the game of StarCraft, approaches to build competent AI for real-time strategy (RTS) games, and the application of evolutionary algorithms to tune AI in RTS games.
A. StarCraft and AI for RTS games

A game of StarCraft consists of two or more opposing players on a predetermined map. Each player constructs and controls buildings and units in real-time. The game is won when a player’s adversaries are defeated (i.e., no enemy buildings remain). This is achieved by destroying the enemy’s forces with military units. Uncertainty over the game state is caused by the fog of war, whereby the units in a region are only visible to the player if the player owns a unit which has sight over the region.

The attention of a player in StarCraft is split towards two aspects of the game: macromanagement and micromanagement. Macromanagement refers to the control of economic activity, and includes subsaspects such as securing bases, resource gathering and troop production. Micromanagement refers to the direct control of individual units and the formulation of tactical plans to attack, defend and harass against the opposing player. Both macromanagement and micromanagement must be handled by the player in real-time. The spontaneity of human players, in conjunction with the need to concurrently handle macro and micro reasonings, establishes the difficulty of developing agents which can rival human players.

The Brood War API (BWAPI) framework\(^1\) was created to enable the development of custom AI agents for StarCraft: Brood War. This is achieved by hacking into the game engine and exposing the control interfaces of the human player to external dynamic-link libraries (DLL) modules loaded by BWAPI. As such, external modules can be written to take control of the human player’s units and play the game in place of the human player.

StarCraft AI tournaments have been held by the research community in recent years to apply and evaluate AI techniques on commercial real-time strategy game platforms. These tournaments may involve the full game of StarCraft, or just a subset (e.g., micromanagement). A listing of the bots which participated in the AIIDE 2010 StarCraft tournament reveals several popular AI techniques, for example, finite state machines (FSM) for macromanagement and/or micromanagement, decision trees and probabilistic inference [6].

To perform competently, an AI agent may require a deep understanding of the game ontology, as well as applying sophisticated reasoning to determine its actions. Many bot authors choose to incorporate their own understanding of the game into the bots directly [7]. Game ontology can also be derived empirically by data mining replays of professional human players [8], [9]. Machine learning algorithms and bayesian inference methods can then be used to predict the strategy of an adversary before it is executed [8], [10]. In addition, Goal-Driven Autonomy has been proposed to enable agents to react in accordance to the dynamic game state [11].

As the state of an RTS game in progress is highly situational, case-based reasoning can be used to select strategies which are applicable to the given situation. A bot must be able to recognise the current situation, and react and adapt accordingly [12]. A bot has a greater chance of success if it is able to predict and plan its actions in advance [1], [13].

B. Evolutionary Algorithms and Application in RTS games

Genetic programming and evolutionary algorithms have been demonstrated to generate and/or optimize human-competitive AI for strategy games. In chess, genetic programming has been demonstrated to generate solutions with competitive performance against expert human-level strategies and CRAFTY, a grandmaster-level chess program [4]. In the real-time strategy game Wargus, evolutionary algorithms have been used to automatically generate game tactics as reported in [5].

In the military research domain, weaknesses in the defences of a force can be discovered by evolving the tactical plans of an invading adversary. Automated-red teaming (i.e., red-teaming automated through evolving agent-based military simulations) has been shown to be able to identify critical weaknesses in military operation plans [14]. Similarly, weaknesses in tactical plans for real-time strategy games may be exposed through the evolution of adversaries.

As reported in [15], optimization through evolutionary computation often involves a wide range of uncertainties which can be divided into four categories: noisy fitness functions; robustness against perturbations in the design parameters or the environment; fitness approximation; and time-varying fitness functions. Noise in fitness evaluations may come from sources such as sensory measurement errors or randomized simulations [15]. To address noise and robustness, it is a common approach to estimate fitness by averaging over a number of samples (i.e., explicit averaging) [15]. When the fitness function evaluation is extremely consuming or a full analytical fitness function is unavailable, approximate fitness functions (also known as meta-models or surrogates) can be used instead [15].

In StarCraft, AI bots have to reason under uncertainty (fog of war) and stochasticity (inherent randomness). Hence, multiple repeated simulations of a candidate solution are necessary to estimate the fitness function accurately. A full StarCraft game involves a myriad of strategic and tactical decisions and may typically take over 20 minutes to complete. Thus it is likely impractical for EC to be used to optimize an AI bot for the whole game. Instead, EC can be used to optimize an agent for a restricted aspect of the game (e.g., micromanagement, build orders) under certain specific scenarios. By placing limitations on the variability of the environment, the variance of the fitness function may be reduced, allowing for a fewer number of simulation replications. Such reductionist approaches may still yield effective solutions.

The optimization of AI bots may require conflicting objectives (i.e., maximise kills while minimising casualties). Multi-objective evolutionary algorithms (MOEA) such as NSGA-II [16] and SPEA2 [17] can be used to generate a set of non-dominated solutions (i.e., in the pareto front). A generic text-based optimization interface such as PISA [18] can be used to easily integrate MOEAs with RTS AI frameworks.

\(^1\)http://code.google.com/p/bwapi/
III. Simulation Framework

This section describes in detail the architecture and implementation of our simulation framework.

A. Overview

The simulation framework is implemented as a BWAPI module, and is used to run simulations of tactical engagement between two opposing forces. Our approach is distinct from the existing StarCraft AI approaches in several ways (e.g., simulation configuration through an external XML file, controlling both forces using the same player).

The objective of the simulation framework is to provide a customisable simulation environment for AI vs. AI conflicts which can be wholly specified through XML, with the results of the simulation saved in an output CSV file. The simulation environment is highly configurable, with the composition and placement of the units and even the AI on each force specified through XML. The AI for each force is loaded from external DLLs as determined by the XML specification. By determining all aspects of a simulation through XML, repeated simulations of tactical scenarios can be performed easily. In this manner, the framework can be easily integrated with evolutionary computation to rapidly find good solutions.

A simulation round involves the setup and execution of a single scenario with two opposing forces–Red and Blue—in tactical conflict, and is completed when an outcome of either Victory, Defeat, or Draw is reached (with respect to Red). In our framework, each force is composed of distinct squads (i.e., groups of individual units). Squad membership is fixed and does not change during execution of a scenario. Each squad has a command queue which is a series of tactical actions to be executed sequentially. Essentially each squad is an independent Finite State Machine (FSM). This force-squad unit organisation allows for a separation of tactical AI (which generates tactical plans to win the scenario), with micro AI (which controls each individual unit to fulfil the current squad command).

Unlike traditional StarCraft gameplay where the two opposing forces are controlled by different players, our framework instead controls both opposing forces under a single player by partitioning the player’s units into two distinct sets of opposing forces, each with their own AI. The units in the two forces are made to engage each other, even though they are actually all owned by the same player. This is possible as StarCraft units belonging to the same player can be made to attack each other (force attack). The primary benefit of this approach is speed. With both forces under the same player, the entire simulation can be executed in single-player mode, and can be sped up beyond the normal speed of StarCraft gameplay. For a simple tactical simulation, this results in a runtime of three seconds as opposed to several minutes in multi-player mode (which is difficult to speed up due to network synchronisation and latency). The game mechanics of tactical combat remain roughly unchanged with two major exceptions: fog-of-war will now have to be emulated by the framework, and automatic unit actions (e.g., automatic engagement of enemy units in combat range, auto-heal ability of the medic unit) will have to be manually executed or suppressed (if desirable or undesirable).

B. Architecture

The framework is implemented in a top-down modular manner as depicted in Figure 1, and consists of the following components:

Simulation Launcher
The launcher loads StarCraft with BWAPI and the framework library.

Core Library
The core library (a BWAPI module) parses the XML specification file, loads the necessary DLL modules, and saves the results into an output file.

Scenario Manager
The Scenario Manager sets up and executes the scenario, terminating when an outcome is reached (i.e., Victory, Defeat or Draw).

AI Managers
Each AI force is governed by three AI managers: the Information Manager, the Squad Manager, and the Micro Manager. Together these AI managers represent the nervous system behind each force.

![Diagram of simulation framework](http://example.com/framework_diagram.png)

Fig. 1. Framework architecture

1) Scenario Manager: The Scenario Manager governs the setup and execution of the simulation model. The setup phase is responsible for creation of the units in each force, allocating the units into squads, and positioning the units at their starting locations. Once all units in the scenario are set, the scenario starts and control of the units is then delegated to the AI managers of the two opposing forces.

During execution of the scenario, the Scenario Manager tracks the status of the squads and the progress towards the simulation objectives. Once an outcome is reached (i.e., Victory, Defeat or Draw), the scenario is terminated and the results of the simulation saved in an output CSV file.

2See http://code.google.com/p/bwapi/wiki/speed for speeding up StarCraft execution
2) Information Manager: In StarCraft, visibility of enemy units is determined by the fog of war. In our framework, the two opposing forces belong to the same player and therefore can see each other. Hence an Information Manager is needed to emulate the fog of war.

The Information Manager periodically determines the set of visible enemy units from the set of all enemy units in the opposing force. This computation is implementation-specific and can be configured to allow for different approaches in computing vision. For instance, an Information Manager may choose to utilise the natural sight range of the units in its computation; or it may simply be "omniscient" by granting vision of all enemy units, sacrificing correctness for speed of computation. Scenarios which are highly dependent on vision (e.g., stealth scenarios) will require an accurate algorithm, while scenarios which are less dependent (e.g., melee combat) can utilise a faster yet inaccurate algorithm which may significantly reduce the runtime for evolutionary computation.

3) Squad Manager: The Squad Manager represents the tactical AI of a force, and determines the high-level commands to be executed by the squads in the force. These high-level commands are specific to our framework, and do not correspond to actual StarCraft commands (which are issued by the Micro Manager). The available command types are listed in Table I.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>Move to a location without engaging nearby enemies.</td>
</tr>
<tr>
<td>Attack-move</td>
<td>Attack-move to a location, engaging nearby enemies.</td>
</tr>
<tr>
<td>Guard</td>
<td>Guard the current location, engaging nearby enemies.</td>
</tr>
<tr>
<td>Sleep</td>
<td>Wait at the current location, without engaging enemies.</td>
</tr>
<tr>
<td>Repeat</td>
<td>Repeat the entire command queue.</td>
</tr>
<tr>
<td>Jump</td>
<td>Jump to the specified command in the queue.</td>
</tr>
</tbody>
</table>

Each squad has a command queue and a pointer to the immediate command within the queue. Once the immediate command is satisfied, the pointer is shifted to the next command. Commands are interpreted and executed by the Micro Manager as described later.

Each squad is essentially modelled as a Finite State Machine executing tactical commands. This allows for a rich possibility of tactical plans. Implementations may determine the commands from the XML specification, or generate commands in real-time to support highly dynamic scenarios such as pursuit-evasion, as depicted in Figure 2.

4) Micro Manager: The Micro Manager issues low-level StarCraft orders to the individual units in a squad so as to fulfil the immediate squad command. For instance, if the squad is commanded to Attack-Move to a location, the Micro Manager may issue the low-level MOVE and ATTACK orders to the individual units to fulfil the command. Micromanagement AI is typically expressed here (e.g., moving in formation, pulling back units with low health). Hence the translation from squad commands given by the Squad Manager to low-level StarCraft orders issued to the individual units may not be straightforward, especially for sophisticated bots.

C. Input XML Specification

All the parameters of a scenario are determined by the XML specification file. The parameters include environment parameters (e.g., map); DLL paths for the managers; squad composition, placement and command queues; and parameters for the AI managers. As the specification language is XML, parameters are not limited to numerical or enumerated values, and can be any valid XML structure/type. The advantage of using XML as the specification language is that it is highly extensible and yet readable by both humans and machines. A simplified example of an XML specification is listed below.

```
<specification>
  <map>TestMap.scx</map>
  <scenario_manager>
    <class>BaseScenarioMgr</class>
  </scenario_manager>
  <forces>
    <red>
      <information_manager>
        <class>OmniscientInformationMgr</class>
      </information_manager>
      <squad_manager>...</squad_manager>
      <micro_manager>...</micro_manager>
      <squad>
        <unit>Protoss Dragoon</unit>
        <command>
          <type>ATTACK_MOVE</type>
          <pos>(x>2000)/x<y>1000</y>/pos>
        </command>
      </squad>
    </red>
    <blue>...</blue>
  </forces>
</specification>
```

D. Output Results CSV File

Results of the simulation are saved in an output CSV file, and includes the scenario outcome, casualties per force and other statistics. This file is utilized by the evolution algorithm for fitness evaluation. This is described in the next section.
IV. Optimizing Scenarios through Evolutionary Computation

Scenario models can be dynamically varied through evolving the XML specification files. In a traditional one-sided evolutionary run, one of the forces is subjected to evolution while the other force remains invariant. This process is performed in the following manner:

1) Define a base XML specification file for the desired scenario.
2) Define the decision variables to be evolved and their boundary values.
3) Define the objective functions. An objective may be minimizing or maximizing, etc.
4) Specify EC settings, such as the evolutionary algorithm (EA) to use, maximum generations, population size and mutation parameters.
5) Generate an initial population of random solutions.
6) Evaluate each candidate solution repeatedly (to account for statistical fluctuations) and assign the candidate’s objective function values from the arithmetic mean of the simulation results.
7) Evolve the individuals using the EA, applying selection, recombination and mutation operators to yield the next generation.
8) Repeat evolutionary loop (i.e., repeat steps 6 and 7) until the termination criteria are satisfied.
9) Return the non-dominated solutions in the last generation.

Evolutionary computation is performed by an external EC module which integrates with the simulation launcher as shown in Figure 3. For our case studies, we have used the Complex Adaptive System Evolver (CASE) framework [19].

V. Case Studies

A. Evolving FreScBot vs. FreScBot

This case study aims to demonstrate the automatic refinement of StarCraft AI bots through evolutionary computation by porting a tournament StarCraft bot, FreScBot, to our framework and evolving it against its original version.

In tournament 1 of the AIIDE 2010 StarCraft AI Competition\(^3\), two opposing bots are squared off in three stages: a melee ground battle involving Zealots (land units with very near attack range), a ranged ground battle involving Dragoons (land units with far attack range), and an aerial battle involving Mutuals (aerial units with near attack range) and Scourges (aerial units with a very near “suicide” attack). As such, contestants have to perform well under varied tactical situations. Each stage begins with both bots controlling identical and opposing units placed in opposite areas of the map and is terminated when one side loses in conflict. FreScBot is a StarCraft bot written by Florent D’Halluin and Valentin Leon-Bonnet, and is the champion of tournaments 1 & 2. It is implemented using multi-agent finite state machines with each unit modelled as a single agent. Its real-valued behavioural parameters (28 in total) are stored in a BehaviourManager class written by the authors. These parameters define certain thresholds (e.g., force ratio to flee) and time intervals used by the bot to make decisions.

We focused on improving FreScBot’s performance in the second stage of Tournament 1 (12 Dragoons vs. 12 Dragoons on a flat terrain) by evolving the parameters in the Behaviour-Manager class. As the evolution parameters are real-valued (as opposed to structural parameters), the extent of improvement is limited especially since the values chosen by the authors were good to begin with (being the champion tournament bot).

1) Porting of FreScBot: Porting of FreScBot into our framework as a Micro Manager involves the creation of an adapter layer between our framework and the original FreScBot classes. This layer is responsible for exposing the FreScBot BehaviourManager’s parameters through the XML specification, and determining owned units and enemy units through our framework’s Information Manager (as both forces are controlled by the same player in our framework).

2) Experiment: In this experiment, each force is composed of a single squad of 12 dragoons, controlled by the ported FreScBot Micro Manager. Information on enemy units is provided by an omniscient Information Manager, which grants vision on all enemy units regardless of distance. This is acceptable as the scenario does not require tactical reasoning. The scenario is completed when one force has been completely wiped out.

12 parameters from the BehaviourManager were selected to evolve (within a range around the original default values) and shown in Table II. These parameters were chosen as they were unlikely to interfere with each other, as highly constrained variables may crash the bot if conflicting values are given. The objectives are to minimise Red casualties while

---

http://eis.ucsc.edu/StarCraftAICompetition

---

\(^3\)http://eis.ucsc.edu/StarCraftAICompetition
maximizing Blue casualties. Five independent evolutionary runs were performed. Each evolutionary run is executed for 30 generations, with a population of 50 candidate solutions per generation. Each candidate solution is repeatedly evaluated 50 times to account for statistical fluctuations. SPEA2 [17] is used as the evolutionary algorithm. To ensure that chirality does not affect the results of the experiment (i.e., bot performs better/worse when it starts at the left side instead of the right side), the starting positions of the forces are alternated every evaluation. This is necessary as certain bots (including FreScBot) may search the map in an asymmetric manner (e.g., left-to-right).

At the end of each evolutionary run, the non-dominated solutions in the final generation are re-evaluated 1000 times to completely account for statistical fluctuations and to derive the success rate of the solutions (i.e., proportion of rounds won). A control experiment consisting of an unevolved Red force vs. an unevolved Blue force is also performed.

3) Results: Each run yielded three to five non-dominated solutions. The success rates of all the non-dominated solutions are depicted in Figure 4. The mean success rate is 58.5%, with five solutions exhibiting significant improvement (Run1PF2, Run4PF1, Run1PF1, Run5PF1, Run5PF5), with Run1PF2 achieving a success rate of 68.3%. The evolved parameters for these five best solutions are shown in Table III.

![Non-dominated solutions for the five runs. The solutions are identified by the run number, and their index in the pareto front (i.e., non-dominated front). The control solution is presented last and has a success rate of 50.4%.

The run times for each evolutionary run are tabulated in Table IV. Each complete run takes approximately three days to complete using a single-core machine (2.8GHz CPU with 3GB RAM) for simulation and evolution.

![TABLE II

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultDefaultPeriod</td>
<td>100</td>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td>defaultFindTargetPeriod</td>
<td>10</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>defaultFightPeriod</td>
<td>10</td>
<td>500</td>
<td>50</td>
</tr>
<tr>
<td>defaultFleePeriod</td>
<td>50</td>
<td>2000</td>
<td>100</td>
</tr>
<tr>
<td>weightOutOfRange</td>
<td>-1000</td>
<td>0</td>
<td>-400</td>
</tr>
<tr>
<td>weightTargetHealth</td>
<td>-500</td>
<td>0</td>
<td>-50</td>
</tr>
<tr>
<td>reevaluateTargetPeriod</td>
<td>100</td>
<td>800</td>
<td>500</td>
</tr>
<tr>
<td>rangeOptimizeMinimumTime</td>
<td>100</td>
<td>2000</td>
<td>500</td>
</tr>
<tr>
<td>forceRadius</td>
<td>250</td>
<td>750</td>
<td>500</td>
</tr>
<tr>
<td>targetRadius</td>
<td>250</td>
<td>750</td>
<td>500</td>
</tr>
<tr>
<td>forceRatioToFlee</td>
<td>0.0</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>hurtSpeedToFlee</td>
<td>0.0</td>
<td>1.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

![TABLE III

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Run1PF2</th>
<th>Run4PF1</th>
<th>Run1PF1</th>
<th>Run5PF1</th>
<th>Run5PF5</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultDefaultPeriod</td>
<td>743</td>
<td>1986</td>
<td>743</td>
<td>562</td>
<td>640</td>
</tr>
<tr>
<td>defaultFindTargetPeriod</td>
<td>444</td>
<td>484</td>
<td>444</td>
<td>436</td>
<td>436</td>
</tr>
<tr>
<td>defaultFightPeriod</td>
<td>238</td>
<td>151</td>
<td>238</td>
<td>60</td>
<td>247</td>
</tr>
<tr>
<td>defaultFleePeriod</td>
<td>81</td>
<td>75</td>
<td>81</td>
<td>53</td>
<td>120</td>
</tr>
<tr>
<td>weightOutOfRange</td>
<td>-125</td>
<td>-493</td>
<td>-873</td>
<td>-318</td>
<td>-314</td>
</tr>
<tr>
<td>weightTargetHealth</td>
<td>-62</td>
<td>-165</td>
<td>-292</td>
<td>-144</td>
<td>-162</td>
</tr>
<tr>
<td>reevaluateTargetPeriod</td>
<td>602</td>
<td>687</td>
<td>602</td>
<td>703</td>
<td>713</td>
</tr>
<tr>
<td>rangeOptimizeMinimumTime</td>
<td>711</td>
<td>844</td>
<td>1396</td>
<td>1968</td>
<td>1995</td>
</tr>
<tr>
<td>forceRadius</td>
<td>257</td>
<td>336</td>
<td>251</td>
<td>258</td>
<td>264</td>
</tr>
<tr>
<td>targetRadius</td>
<td>588</td>
<td>450</td>
<td>443</td>
<td>461</td>
<td>721</td>
</tr>
<tr>
<td>forceRatioToFlee</td>
<td>0.126</td>
<td>0.431</td>
<td>0.079</td>
<td>0.140</td>
<td>0.133</td>
</tr>
<tr>
<td>hurtSpeedToFlee</td>
<td>0.269</td>
<td>0.260</td>
<td>0.269</td>
<td>0.192</td>
<td>0.392</td>
</tr>
</tbody>
</table>

![TABLE IV

<table>
<thead>
<tr>
<th>Run</th>
<th>Total time (h)</th>
<th>Time per generation (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run1</td>
<td>70.4</td>
<td>2.27</td>
</tr>
<tr>
<td>Run2</td>
<td>69.9</td>
<td>2.25</td>
</tr>
<tr>
<td>Run3</td>
<td>74.9</td>
<td>2.41</td>
</tr>
<tr>
<td>Run4</td>
<td>82.8</td>
<td>2.67</td>
</tr>
<tr>
<td>Run5</td>
<td>70.5</td>
<td>2.27</td>
</tr>
</tbody>
</table>

![TABLE V

This case study demonstrates that automatic refinement of parameters is sufficient to provide significant improvements in FreScBot. However, this test scenario involves only a single type of unit (12 Dragoons) in a symmetric engagement on a flat terrain against a fixed enemy (unevolved FreScBot). It remains to be seen if a robust improvement to FreScBot can be attained by simply tuning its parameters. Also it would be interesting to evaluate the evolved FreScBot against the other AIIDE tournament bots and verify that it still yields superior performances.

B. Generating attack path to maximise enemy casualties

A common tactical problem in strategy games is to formulate an effective attack path to cause maximal damage, while avoiding critical defensive positions (i.e., to increase survivability). This case study aims to demonstrate that our framework can be used to model and evolve such tactical plans in StarCraft.

1) Experiment: In this simple scenario, the Red force of 16 Dragoons navigates a variable attack path to engage the Blue squads scattered around the map. The Blue force is composed of four light squads which can be easily killed, and two squads of Battlecruisers which are impossible to defeat. The scenario is further elaborated in Figure 5.
Fig. 5. Map of the incursion scenario. Red’s 16 Dragoons at the bottom-left corner follow an attack path of 4 waypoints generated through evolution, engaging Blue’s light squads scattered around the map and two groups of enemy Battlecruisers which should be avoided. Finally, at the end of the attack path, Red moves to the centre of the map to terminate the scenario by engaging the Battlecruisers there.

Fog of war on both sides is emulated by the Information Manager. Blue squads do not exchange vision of enemy units with each other as otherwise all Blue squads will move to engage as soon as one Red squad comes into proximity of any Blue squad. Micromanagement of the units is performed on both sides by a simple Micro Manager implementation which engages the enemy by strictly targeting the closest enemy units first.

The objective of the scenario is for Red to maximise the enemy’s casualties whilst minimizing the total length of the attack path. Five independent evolutionary runs were performed to optimize the four variable waypoints (excluding the start and end waypoints). Each evolutionary run is executed for 30 generations, with a population of 50 candidate solutions per generation. Each candidate solution is repeatedly evaluated 30 times to account for statistical fluctuations. SPEA2 [17] is used as the evolutionary algorithm.

2) Results: The best solution for each run is determined by the candidate which inflicted the most enemy casualties in the final generation, and are shown in Table V.

Most solutions were able to kill off all the light squads (i.e., Blue Casualties $\geq 32$). The attack paths generated by these solutions are depicted in Figure 6.

The run times for each evolutionary run are tabulated in Table VI. Each complete run takes approximately 6.5 days to complete using a single-core machine (2.8GHz CPU with 3GB RAM) for simulation and evolution.

These results demonstrate that tactical plans can be modelled through the framework and optimized using evolutionary computation.

VI. CONCLUSION AND FUTURE WORK

We presented our framework in which AI bots for StarCraft can be modularly constructed and evolved. We demonstrated in our first case study that evolutionary computation (EC) can be used to optimize AI bots; and in our second case study that tactical plans can be modelled and evolved. In each case study, the evolutionary run took less than one week to complete on a single computer. Our case studies represent a preliminary investigation into the applicability of EC with respect to optimization of StarCraft AI bots and were highly simplified to facilitate analysis.

The framework is necessarily limited as it can only model tactical combat and ignores concepts such as economy and base production. However, like in chess, studying small scale tactical setups may provide valuable insights into playing a competent game of StarCraft.

The choice of XML as the specification language allows bot authors to specify not only real-valued parameters, but...
also structural parameters. For instance, a Micro Manager may determine its rules through XML structures. These XML structures can then be varied through genetic programming techniques, allowing for highly expressive AI.

Although in our study we focused on the evolution of combat tactical AI, EC can also be used to generate or refine real-time strategy AI in other aspects such as:

1) Generation of base layouts with regard to mobility, access to resources and defensibility
2) Strategy selection (i.e., assigning preferential weights to strategies)

The solutions generated by EC may be optimal for a specific scenario yet difficult to generalise to the full game. Indeed it may turn out that a single optimal strategy does not exist even for a restricted aspect of the game. A bot’s performance is then largely determined by its ability to select solutions which are most applicable to the current situation. EC generated solutions to specific situations can be used to populate a knowledge base closely approximating the problem to optimize. This may significantly reduce the computational cost of an evolutionary run, and only then apply the evolved solutions.

An alternative approach to simulating whole RTS scenarios for fitness evaluation is to construct a meta-model which closely approximates the problem to optimize. This may significantly reduce the computational cost of an evolutionary run, allowing for EC to be used during a game in progress. This approach is especially suited when the problem is analytically difficult to solve. For instance, an abstract model of the armies and bases in the game can be used to generate and evolve tactical plans in real time which can maximise damage to the enemy while avoiding critical defensive positions. As EC is a stochastic search-based approach, creative strategies may even be discovered and evaluated during the course of a game.

ACKNOWLEDGMENT
We would like to thank the Defence Research and Technology Office, Ministry of Defence, Singapore, for sponsoring the Evolutionary Computing Based Methodologies for Modeling, Simulation and Analysis project.

REFERENCES