Building Placement Optimization in Real-Time Strategy Games

Nicolas A. Barriga, Marius Stanescu, and Michael Buro
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{barriga|astanesc|mburo}@ualberta.ca

Abstract

In this paper we propose using a Genetic Algorithm to optimize the placement of buildings in Real-Time Strategy games. Candidate solutions are evaluated by running base assault simulations. We present experimental results in SparCraft — a StarCraft combat simulator — using battle setups extracted from human and bot StarCraft games. We show that our system is able to turn base assaults that are losses for the defenders into wins, as well as reduce the number of surviving attackers. Performance is heavily dependent on the quality of the prediction of the attacker army composition used for training, and its similarity to the army used for evaluation. These results apply to both human and bot games.

Introduction

Real-Time Strategy (RTS) games are fast-paced war-simulation games which first appeared in the 1990s and enjoy great popularity ever since. RTS games pose a multitude of challenges to AI research:

- RTS games are played in real-time — by which we mean that player actions are accepted by the game engine several times per second and that game simulation proceeds even if some players elect not to act. Thus, fast to compute but non-optimal strategies may outperform optimal but compute-intensive strategies.
- RTS games are played on large maps on which large numbers of units move around under player control — collecting resources, constructing buildings, scouting, and attacking opponents with the goal of destroying all enemy buildings. This renders traditional full-width search infeasible.
- To complicate things even further, most RTS games feature the so-called "fog-of-war", whereby players' vision is limited to areas around units under their control. RTS games are therefore large-scale imperfect information games.

The initial call for AI research in RTS games (Buro 2004) motivated working on RTS game AI by describing the research challenges and the great gap between human and computer playing abilities, arguing that in order to close it classic search algorithms will not suffice and proper state and action abstractions need to be developed. To this day, RTS game AI systems are still much weaker than the top human players. However, the progress achieved since the original call for research — recently surveyed in (Ontañón et al. 2013) — is promising. The main research thrust so far has been on tackling sub-problems such as build-order optimization, small-scale combat, and state and action inference based on analysing thousands of game transcripts. The hope is to combine these modules with high-level search to ultimately construct players able to defeat strong human players. In this paper we consider the important problem of building placement in RTS games which is concerned with constructing buildings at strategic locations with the goal of slowing down potential enemy attacks as much as possible while still allowing friendly units to move around freely.

Human expert players use optimized base layouts, whereas current programs do not and therefore become prone to devastating base attacks. For example, Figure 1 shows a base that is laid-out well by a human player, while Figure 2 depicts a rather awkward layout generated by a top StarCraft bot. The procedure we propose here assumes a given build-order and improves building locations by means of maximizing survival-related scores when being exposed to simulated attack waves whose composition has been learned from game transcripts.

In what follows we will first motivate the building placement problem further and discuss related literature. We then present our algorithm, evaluate it empirically, and finish the paper with concluding remarks and ideas for future work.

Background

Strategic building placement is crucial for top-level play in RTS games. Especially in the opening phase, player's own bases need to be protected against invasions by creating wall-like structures that slow opponents down so that they cannot reach resource mining workers or destroy crucial buildings. At the same time, building configurations that constrain movement of friendly units too much must be avoided. Finding good building locations is difficult. It involves both spatial and temporal reasoning, and ranges from blocking melee units completely (Certicky 2013) to creating bottlenecks or even maze-like configurations that maximize the time invading units are exposed to own static and mobile

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Figure 1: Good building placement. Structures are tightly packed and supported by cannons. (Screenshot taken from a Protoss base layout thread in the StarCraft strategy forum on TeamLiquid[1])

Figure 2: Weak building placement: structures are scattered and not well protected. (Screenshot taken from a match played by Skynet and Aiur in the 2013 AIIDE StarCraft AI competition (Churchill 2013a).

defenses. Important factors for particular placements are terrain features (such as ramps and the distance to expansion locations), the cost of constructing static defenses, and the type of enemy units.

Human expert players are able to optimize building locations by applying general principles such as creating chokepoints, and then refining placement in the course of playing the same maps over and over and analyzing how to counter experienced opponent attacks. Methods used in state-of-the-art RTS bots are far less sophisticated (Ontanón et al. 2013). Some programs utilize terrain analysis library BWTA (Perkins 2010) to identify chokepoints and regions to decide where to place defenses. Others simply execute pre-defined building placement scripts program authors have devised for specific maps. Still others use simple-minded spiral search around main structures to find suitable building locations. In contrast, our method — that will be described in the next section in detail — combines fast combat simulation for gauging building placement quality with data gathered from human and bot replays for attack force estimation and stochastic hillclimbing for improving placements. The end result is a system that requires little domain knowledge and is quite flexible because the optimization is driven by an easily adjustable objective function and simulations rather than depending on hard-coded domain rules — described for instance in (Certicky 2013).

[1] http://www.teamliquid.net/forum/bw-strategy/64136-protoss-base-layout

Building placement is a complex combinatorial optimization problem which can't be solved by exhaustive enumeration on today's computers. Instead, we need to resort to approximation algorithms such as simulated annealing, tabu search, and Genetic Algorithms — which allow us to improve solutions locally in an iterative fashion. In this paper we opt for Genetic Algorithms because building placement problem solutions can be easily mapped into chromosomes and mutation and crossover operations are intuitive and can be implemented efficiently. Good introductions to the subject can be found in (Mitchell 1998) and (Goldberg 1989). For the purpose of understanding our building placement algorithm it suffices to know that Genetic Algorithms

- are stochastic hill-climbing procedures that
- encode solution instances into objects called chromosomes,
- maintain a pool of chromosomes which initially can be populated randomly or biased towards good initial solutions,
- generate new generations of chromosomes by random mutations and using so-called crossover operations that take two parents based on their fitness to generate offspring,
- and in each iteration remove weak performers from the chromosome pool.

Genetic Algorithms have been applied to other RTS game AI sub-problems such as unit micro-management (Liu, Louis, and Nicolescu 2013), map generation (Togelius et al. 2010), and build-order optimization (Köstler and Gmeiner 2013).

Algorithm and Implementation

Our Genetic Algorithm (GA) takes a battle specification (described in the next paragraph) as input and optimizes the building placement. To asses the quality of a building placement we simulate battles defined by the candidate building placements and the mobile units listed in the battle specification. Our simulations are based on fast scripted unit behaviors which implement basic combat micro-management, such as moving towards an enemy when a unit is getting shot but doesn't have the attack range to shoot back, and smart No-OverKill (NOK) (Churchill, Saffidine, and Buro 2012) targeting. The defending player tries to stay close to his buildings to protect them, while the attacker tries to destroy buildings if he is not attacked, or kill the defender units

otherwise. Probes — which are essential for the economy — and pylons — needed for supply and for enabling other buildings — are considered high-priority targets. Retreat is not an option for the attacker in our simulation because we are interested in testing the base layout against a determined attack.

Our GA takes a battle specification from a file. This input consists of starting positions and initial health points of all units, and the frame in which each unit joined the battle to simulate reinforcements. The units are split between the defender player, who has buildings and some mobile units, and the attacking player who does not have any buildings.

This data can be obtained from assaults that happened in real games — as we do in our experiments — or it could be created by a bot by listing the buildings it intends to construct, units it plans to train, and its best guess for the composition and attack times of the enemy force.

Starting from a file describing the battle setup, a genome is created with the buildings positions to be optimized. Fixed buildings, such as a Nexus or Assimilator, and mobile units are stored separately because their positions are not subject to optimization.

We implemented our GA in C++ using GAlib 2.4.7 (Wall 2007) and SparCraft (Churchill 2013b). GAlib is C++ library that provides the basic infrastructure needed for implementing GAs. SparCraft is a StarCraft combat simulator. We adapted the version from (Churchill 2013b) by adding required extra functionality such as support for buildings, the ability to add units late during a battle, and basic collision tests and path-finding. In this implementation, all building locations are impassable to units and thus constrain their paths, and also possess hit points, making them a target for enemy units. The only buildings which have extra functionality are static defenses such as Protoss Photon Cannons, which can attack other units, and Pylons, which are needed to power most Protoss buildings. StarCraft (Blizzard Entertainment 1998), one of the most successful RTS games ever, has become the de-facto testbed for AI research in RTS games after a C++ library was released in 2009 that allows C++ programs to interact with the game engine to play games (Heinermann 2014).

Genetic Algorithm

Our GA is a generational Genetic Algorithm with non-overlapping populations and elitism of one individual. This is the *simple* Genetic Algorithm described in (Goldberg 1989), which in every generation creates an entirely new population of descendants. The best individual from the previous generation is copied over to the new population by elitism, to preserve the best solution found so far. We use roulette wheel selection with linear scaling of the fitness. Under this scheme, the probability of an individual to get selected is proportional to its fitness. The termination condition is a fixed number of generations.

Genetic Representation

Each gene contains the position of a building, and an individual's chromosome is a fixed size array of genes. Order is always maintained (e.g., i-th gene always corresponds to the i-th building) to be able to relate each gene to a specific building.

Initialization

From a population of N individuals, the first $N-1$ individuals are initialized by randomly placing the buildings in the area originally occupied by the defender's base. A repair function is then called to fix illegal building locations by looking for legal positions moving along an outward spiral. Finally, we seed the population (the N-th individual) with the actual building locations from the battle description file. Using real layouts taken from human games is a feasible strategy, not only for our experimental scenario, but for a real bot competing in a tournament. Major tournaments are played on well known maps, for which we have access to at least several hundreds game replays each, and it is highly likely that some of those use a similar building composition as our bot. Otherwise, we can use layout information from our own (or another) bot's previous games, which we later show to produce similar results.

Genetic Operators

Because the order of the buildings in the chromosome does not hold any special meaning, there is no "real-world" relationship between two consecutive genes which allows us to use *uniform crossover* rather than the more traditional N-point crossover. Each crossover operation takes two parents and produces two children by flipping a coin for each gene to decide which child gets which parent's gene. Afterwards, the same repairing routine that is used in the initialization phase is applied if the resulting individuals are illegal.

For each gene in a chromosome, the *mutation operator* with a small probability will move the associated building randomly in the vicinity of its current location. The vicinity size is a configurable parameter that was set to a 5 build tile radius for our experiments.

Fitness Function

Fitness functions evaluate and assign scores to each chromosome. The higher the fitness, the better solution the chromosome represents. To compute this value, we use SparCraft to simulate battles between waves of attackers and the individual's buildings plus mobile defenders. After the battle is concluded, we use the value (negative if the attackers won) of the winner's remaining army as the fitness score. For a given army, we compute its *value* using the following simple rules, created by the authors based on their StarCraft knowledge:

- the value of an undamaged unit is the sum of its mineral cost and 1.5 times its gas cost (gas is more difficult to acquire in StarCraft)
- the value of a damaged unit is proportional to its remaining health (e.g., half the health, half the value)
- values of workers are multiplied by 2 (workers are cheap to produce but very important)
- values of pylons are multiplied by 3 (buildings cannot function without them, and they increase the supply limit)

- finally, the value of the army is the sum of the values of its units and structures.

If we are simulating more than one attacker wave, the fitness is chosen as the lowest score after simulating all battles. Preliminary experiments showed that this gives a more robust building placement than taking the average over all battles.

Evaluation

We are interested in improving the building placement for StarCraft scenarios that are likely to be encountered in games involving highly skilled players. In particular, we focus on improving the buildings' location for given build orders and probable enemy attack groups. To obtain this data, we decided to use both high-level human replays (Synnaeve and Bessiere 2012) and replays from the top-3 bots in the last AIIDE StarCraft competition (Churchill 2013a).

For a given replay, we first parse it and identify all base attacks, which are defined as a group of units attacking at least one building close to the other player's base. A base attack ends when the game is finished (if one player is completely eliminated) or when there was no unit attack in the last 15 seconds. We save all units present in the game during such a battle in a Boolean "adjacency" matrix A, where two units are adjacent if one attacked the other one or if at some point they were very close to each other during this battle interval (this matrix is symmetric). By multiplying this matrix repeatedly we can compute an "influence" matrix (e.g., A^2 tells us that a unit X *influenced* a unit Z if it attacked or was close to a unit Y that attacked or was close to Z). From this matrix we can read the connected components — in which any two units are connected to each other by paths — and thus we can easily separate different battles across the map and extract base assaults. We then filter or fix these battles according to several criteria, such as the defending base containing at least three buildings, and both players having only unit types that are supported by SparCraft. Another limitation is that SparCraft implements fast but inaccurate path-finding and collisions, so in a small percentage of games units can get "stuck". We eliminate these games from our data analysis, and thus can end up with different number of battles in different experiments. At this point the Protoss faction is the one with the most features supported by the simulator. We therefore focus on Protoss vs. Protoss battles in this paper. After considering all these restrictions, 57 battles from human replays and 31 from bot replays match our criteria.

To avoid having too few examples for each experiment, we ran the GA over this dataset several (2 to 4 depending on the experiment) times. The results vary because GA is a stochastic algorithm.

Each base assault has a fixed (observed) build order for the defending player and a list of attacking enemy units, which can appear at different time points during the battle. Using the GA presented in the previous section we try to improve the building placement and then simulate the base attack with SparCraft to test the new building configuration.

We found that most extracted base assaults strongly favour one of the players. In real games either the attacker has just a few units and tries to scout and destroy a few buildings and then retreat, or he already defeated most defender units in a battle that was not a base assault and then uses his material superiority to destroy the enemy base. These instances are not very useful test cases because if the army compositions are too unbalanced, the building placement is of little importance. Consequently, to make building placement relevant, we decided to transform all games into a win for the attacker (i.e., destroying all defender units) by adding basic melee units (zealots), while keeping the build order unchanged. We believe this is the least disruptive way of balancing the armies. Additionally, it allows us to show statistics on the percentage of games lost by the defender that can be turned into a win (i.e, destroying all attacker units and keeping some buildings alive) by means of smarter building placement.

Figure 3: Bad building placement (created manually). The red arrow indicates the attack trajectory.

Figure 4: Typical improved building placement.

An example of a less than optimal initial Protoss base layout is shown in Figure 3. Figure 4 shows an improved layout for the same buildings, proposed by our algorithm. The attacking units follow the trajectory depicted by the red arrow.

For every base assault the algorithm performs simulations using the defender buildings and army described in the battle specification file and attack groups from all other base assaults extracted from replays on the same map and

close in time to the original base attack used for training. These attack waves try to emulate the estimate a bot could have about the composition of the enemy's army, having previously played against that opponent several times. After the GA optimizes the configuration the final layout is tested against the original attack group (which was not used for training/optimizing). We call this configuration **cross-optimized**.

As a benchmark we also run the GA optimization against the attacker army that appeared in the actual game that we actually use for testing. This provides an upper bound estimate on the improvement we could obtain in a real game if we had a perfect estimate of the enemy's army. We call this configuration **direct-optimized**.

All experiments were performed on an Intel(R) Core2 Duo P8400 CPU 2.26GHz with 4 GB RAM running Fedora 20. The software was implemented in C++ and compiled with g++ 4.8.2 using -O3 optimization.

Results

Figure 5 shows the improvements obtained by the GA using a population of 15 individuals and evolving for 40 generations. This might seem too little for a regular GA, but it is necessary due to the time it takes to run a simulation, and as the results show, it is sufficient. The cross-optimized GA manages to turn about a third of the battles into wins, while killing about 3% more attackers. If it had perfect knowledge of what exact attack wave to expect, represented by the direct-optimized GA, it could turn about two thirds of the battles into wins while killing about 19% more attackers.

Work on predicting the enemy's army composition would prove very valuable when combined with our algorithm. However, we are not aware of any such work, except for some geared toward identifying general strategies (Weber and Mateas 2009) or build orders (Synnaeve and Bessière 2011).

Figure 6 compares results obtained optimizing human and bot building placements. There is some indication that bot building placement gains more from using our algorithm as more attackers are killed after optimization. However, it seems that the advantage gained is not enough to turn more defeats into victories. This result might be explained by the fact that we do not directly compare human and bot building placements, as the base assaults are always balanced such as the attacker wins before optimization. This takes away any advantage the human base layout might initially hold over ones that bots create.

Figure 7 shows that running longer experiments with a larger population and more generations leads to better results, as expected. A bot with a small number of pre-set build orders having access to some of his opponent's past games could improve its building placement by optimizing the building placements offline. Decent estimates for the attacking armies could be extracted from the replays and the bot could then use bigger GA parameters to obtain better placements because time is not an issue for offline training. However, Figure 5 shows that the best scores are attained by using accurate predictions of the enemy's army — which are more likely to be obtained during a game rather than from past replays — indicating that another possible promising approach is to use a small and fast in-game optimizer seeded with the solution from a big and slow offline optimization.

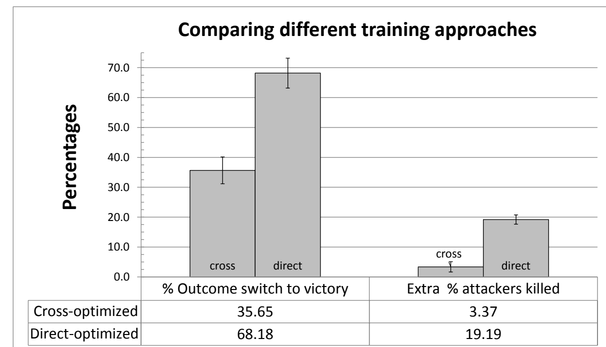

Figure 5: Percentage of losses turned into wins and extra attackers killed when cross-optimizing and direct-optimizing. 115 cross-optimized and 88 direct-optimized battles were played . Error bars show one standard error.

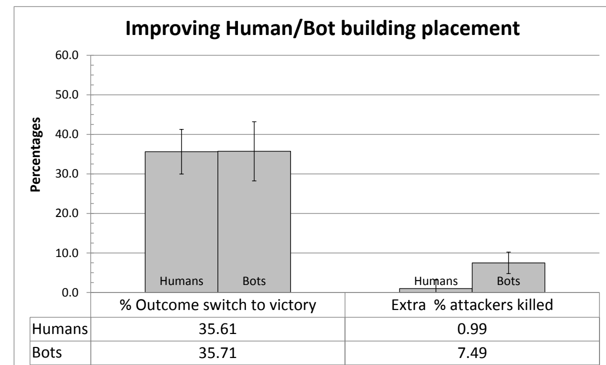

Figure 6: Percentage of losses turned into wins and extra attackers killed when cross-optimizing, comparing results for human and bot data. 106 human battles and 52 bot battles were played. Error bars show one standard error.

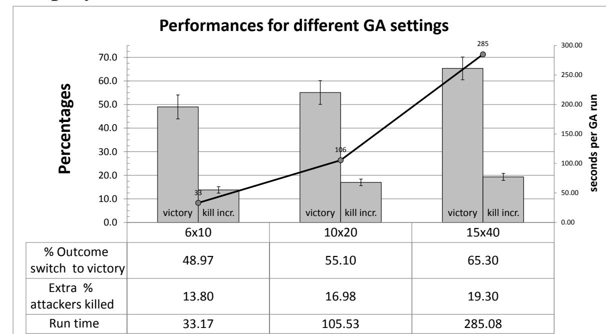

Figure 7: Percentage of losses turned into wins, extra attackers killed, and average run time for three direct-optimizing GA settings. We compare populations of 6, 10 and 15 individuals, running for 10, 20 and 40 generations respectively. 98 battles were played for each configuration. Error bars show one standard error.

Conclusions and Future Work

We have presented a technique for optimizing building placement in RTS games that applied to StarCraft is able to help the defending player to better survive base assaults. In our experiments between a third and two thirds of the losing games are turned into wins, and even if the defender still loses games, the number of surviving attackers is reduced by 3% to almost 20% depending on our ability to estimate the attacker force. The system's performance is highly dependent on how good this estimate is, inviting some research in the area of opponent army prediction.

The proposed algorithm can easily accommodate different maps and initial building placements. We ran experiments using over 20 StarCraft maps, and base layouts taken from both human and bot games. Bot games show a slightly larger improvement after optimization, as expected. Using simulations instead of handcrafted evaluation functions ensures that this technique can be easily ported to other RTS games for which simulators are available.

We see three avenues for extending our work:

- extending the application,
- enhancing the simulations, and
- improving the optimization algorithm itself.

The application can be naturally extended by including the algorithm in a bot to optimize preset build orders against enemy armies extracted from previous replays against an enemy we expect to meet in the future. When a game starts, the closest one to our needs can be loaded and used either as is, or as a seed for online optimization. Exclusive offline optimization can work because bots don't usually perform a wide variety of build orders. Online, at roughly 30 seconds per run, can be done as long as the bot has a way of predicting the most likely enemy army.

Another possible extension is to add functionality for training against successive attack waves, arriving at different times during the build order execution. The algorithm would optimize the base layout until the first attack wave, and then consider all previous buildings as fixed. Until the next attack wave arrives, it would optimize only the positions of the buildings to be constructed. The fitness function would take into account the scores for all successive waves.

The simulations could be greatly enhanced by adding support for more advanced unit types and game mechanics, such as bunkers, flying units, spell-casters and cloaking. This would allow us to explore Terran and Zerg building placements in StarCraft, at any point in the game.

Finally, the algorithm could benefit from exploring different ways of combining the evaluation of attack waves into the fitness function. Currently the fitness is the lowest score obtained after simulating all attack waves, which led to better results than using the average. The GA could also benefit from the use of more informed operators which integrate domain knowledge, and are aware of choke-points, how to build "walls", or how to protect the worker line.

References

Blizzard Entertainment. 1998. StarCraft: Brood War. http://us.blizzard.com/en-us/games/sc/.

Buro, M. 2004. Call for AI research in RTS games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, 139–142.

Certicky, M. 2013. Implementing a wall-in building placement in StarCraft with declarative programming. *arXiv preprint arXiv:1306.4460*.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*.

Churchill, D. 2013a. 2013 AIIDE StarCraft AI competition report. http://www.cs.ualberta.ca/ cdavid/starcraftaicomp/report2013.shtml.

Churchill, D. 2013b. SparCraft: open source StarCraft combat simulation. http://code.google.com/p/sparcraft/.

Goldberg, D. E. 1989. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Professional.

Heinermann, A. 2014. Broodwar API. http://code.google.com/p/bwapi/.

Köstler, H., and Gmeiner, B. 2013. A multi-objective genetic algorithm for build order optimization in StarCraft II. *KI-Künstliche Intelligenz* 27(3):221–233.

Liu, S.; Louis, S. J.; and Nicolescu, M. 2013. Using CIGAR for finding effective group behaviors in RTS games. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8. IEEE.

Mitchell, M. 1998. *An introduction to genetic algorithms*. MIT press.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *TCIAIG* 5(4):293–311.

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. *Artificial Intelligence* 168–173.

Synnaeve, G., and Bessière, P. 2011. A Bayesian model for plan recognition in RTS games applied to StarCraft. In AAAI., ed., *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2011)*, Proceedings of AIIDE, 79–84.

Synnaeve, G., and Bessiere, P. 2012. A dataset for StarCraft AI & an example of armies clustering. In *AIIDE Workshop on AI in Adversarial Real-time games 2012*.

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; and Yannakakis, G. N. 2010. Multiobjective exploration of the StarCraft map space. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 265–272. IEEE.

Wall, M. 2007. GAlib: A C++ library of genetic algorithm components. http://lancet.mit.edu/ga/GAlib.html.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. In *IEEE Symposium on Computational Intelligence and Games (CIG)*.