Comparing Heuristic Search Methods for Finding Effective Real-Time Strategy Game Plans

Christopher Ballinger and Sushil Louis
University of Nevada, Reno
Reno, Nevada 89503
{caballinger, sushil}@cse.unr.edu

***Abstract*—This paper compares genetic algorithms against bit-setting hill-climbers for generating competitive plans to beat an opponent in the initial stages of real-time strategy games. Specifically, we search for build orders that generate the right mix of entities and attack orders and compare the algorithms' performance against optimal plans from exhaustive search. Since multiple possible global optima exist, three hand-coded opponents that follow different strategies serve to provide a baseline for plan comparisons. Our results show that while our hill-climber takes three hours to produce optimal plans against our three hard-coded baselines, it only finds these plans six percent of the time. On the other hand, genetic algorithms routinely find the best plans against our baselines but take significantly longer. This work helps game AI designers evaluate the strengths of these types of heuristic search algorithms and serves to inform our research on improving evolutionary approaches to RTS game player design.**

I. Introduction

Finding effective strategies in Real-Time Strategy (RTS) games presents a challenging problem. RTS game players must compete for resources, build up an economy that is able to support their military force, expand their control over the map, and eventually destroy their opponents base. These are military themed games and any advances in developing RTS game players will impact planning and execution in command and control for military assets and, more immediately, impact military training simulations through the development of smart, realistic opponents.

Several problems make finding effective strategies complicated. Many types of units are available, each with their own costs and dependencies. For example, in StarCraft, building a *marine* requires 50 minerals and takes 25 seconds to build. However, before you can start building *marines* you must build at least one *barracks*, which requires 150 minerals and takes 80 seconds to build. Choosing which units to build and the order in which to build them leads to a combinatorially explosive number of possible strategies. An exhaustive search though strategy space is generally not feasible and in this paper, we apply and compare genetic algorithms (GAs) and a bit-setting hill-climber for finding effective strategies.

Our overall goal is to create competent RTS game players and this paper compares Genetic Algorithms (GAs) with Bit-Setting Hill-Climbers (HCs) - two approaches to searching the space of possible strategies for good game playing strategies. However, several issues make such comparisons difficult. First, there is no optimal strategy. RTS games are designed like rock-paper-scissors games and for every possible good strategy there is a counter strategy that can beat it. This means that it is not trivial to compare two strategies because strategy S_1 beating S_2 does not imply that S_1 is better than S_2 since S_1 could be beaten by S_3 which could be beatable by S_2, as shown by Figure 1. Furthermore, S_2 could beat a larger number of strategies and thus be more robust than S_1. To deal with these issue, we created three very different hand coded strategies and compare how strategies generated by GAs and hill-climbers do against these three baselines.

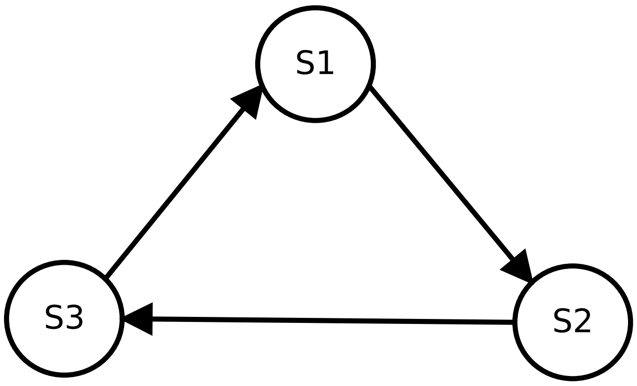

Fig. 1. Three strategies that defeat each other.

Once we recognize and overcome the difficulties in comparing strategies, we want to see how good our two algorithms do in searching strategy spaces. Do they find good strategies, strategies that beat one or more of our three baselines? How long does it take? How often are good strategies found (are GAs more robust than hill-climbers?). In order to make such comparisons more meaningful, we also need to look at the space of all possible strategies, that is, we need to exhaustively list all possible strategies and evaluate their performance against our three baselines. We therefore restrict our search space enough to make it tractable to exhaustively list and evaluate all strategies on our 400 core cluster. We can then compare GAs and hill-climbers with the results of exhaustive search to answer questions of solution quality, robustness, and search time complexity.

This paper answers the questions posed in the previous paragraph with respect to our baselines. In addition, we also describe and qualitatively analyze the properties of strategies found by the two search algorithms. Our results show that hill-climbing is capable of producing effective strategies after three hours, but only finds effective solutions 6% of the time out of thirty-two runs. Our GA took twenty hours to run, but the entire population of individuals converged to three unique optimal solutions 100% of the time out of ten runs. Given enough time, GAs thus appear to be better than our particular hill-climber in generating high quality strategies. However, if we are more interested in finding a quick solution, we may opt for a hill-climber. These trade-offs inform current research in developing a full game AI for WaterCraft, our open-source RTS clone of StarCraft.

The next section describes related work in generating build-orders, a sequence of commands, in RTS games and RTS simulation environments. In Section III we describe our representation and the details of our hill-climber and GA implementations. Section IV presents preliminary results and shows a comparison of solutions produced by our methods. Finally, the last section provides conclusions and discusses possible future work.

II. Related Work

Much work has been done in computational and artificial intelligence approaches to designing RTS game players. One approach uses case based planning, taking a database of expert demonstrations and modifying them as similar problems are encountered. Ontañón created one example of such a system which was used to quickly produce a good game player (game AI) that wins often [1]. However, such a system not only requires recording the actions of an expert that played the game, but also having the expert "explain" to the system what goal was being accomplished with each action taken. Case-injected GAs [2] are similar to case-based planning. Miles and Louis used a case-injected GA to evolve a RTS player. Unlike case-based planning, a case-injected GA does not require a case-base from the start, because the GA will learn new cases as the GA works on the problem. Using case-injection increased the rate at which the GA learned new strategies [3]. Avery and Louis also did work on co-evolving team tactics using influence maps, allowing a group of entities to adapt to opponent moves [4]. Tsapanos used a zeroth-level classifier, which involved a GA and reinforcement learning. The GA in this case found new rules that helped reinforcement learning algorithms find solutions [5]. Spronck used a method called dynamic scripting to create game AI for a non-RTS game. Dynamic scripting involves selecting rules from a knowledge base with a certain probability for each individual unit that the game spawns. The probability of a rule being selecting during this process was set using reinforcement learning as well. Initially, Spronck populated the knowledge base with hand coded rules [6]. However, Spronck et. al. later used a GA to generate rules for the dynamic scripting knowledge base, which further improved their results [7].

While these previous studies focused on the development of complete RTS game players that manage all aspects of RTS game play, we focus solely on creating build-orders. A build-order is simply a sequence of commands that results in a particular set of units and buildings (the build). Ordering is important because of pre-requisite relationships between units and buildings.

Case-based reasoning has also been applied to build-orders, as opposed to a complete game [8]. A depth first branch and bound algorithm was used by Buro to adapt a build-order in real time [9]. Tong used a GA to tune an artificial neural network (ANN) for determining what units the ANN should use to counter the opponents attack force [10]. Sandberg used a similar approach, but used the GA to tune potential field parameters for small-scale combat [11]. Others work in developing some aspect of game AI for RTS games can be found in [12], [13], [14], [15], [16]. In addition to the variety of RTS game computational and artificial intelligence approaches, there are also a variety of RTS environments in which game computational and artificial intelligence can be developed [17], [18], [19], [20], [21].

We developed WaterCraft for researching evolutionary algorithms in RTS games and wrote Watercraft primarily in Python with some C/C++ to speed up physics. WaterCraft uses the popular Python-OGRE graphics engine for the GUI and graphics display [22]. We modeled the game play in WaterCraft around the popular commercial game, StarCraft [23]. While Watercraft lacks some features provided in commercial games, we have implemented some of the core features of all RTS games. Players can build several types of buildings and units, with the objective of destroying their opponents base. WaterCraft's graphical user interface (GUI) resembles and functions similar to the player GUI in other RTS games. The player also has the option of playing against the AI or another player over the network.

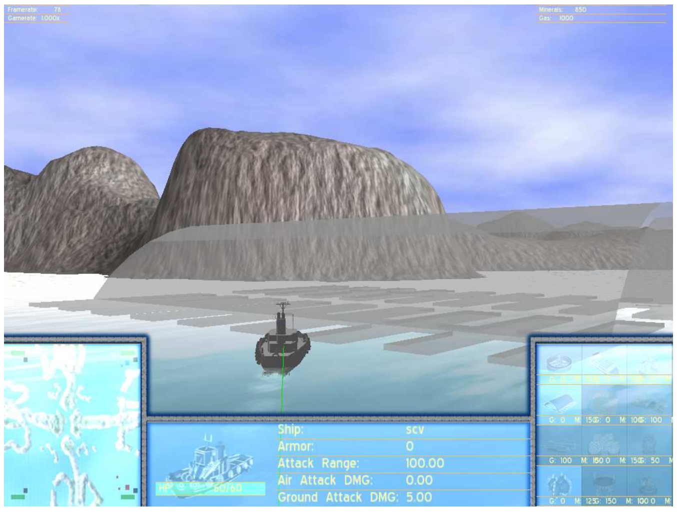
Framerate: 78
Gamerate: 1.000x
Minerals: 850
Gas: 1000
Ship: scv
Armor: 0
Attack Range: 100.00
Air Attack DMG: 0.00
Ground Attack DMG: 5.00
HP 60/60

Fig. 2. An "SCV" building a barracks in WaterCraft.

Other similar projects have been used in the past for research in RTS games. The Brood-War API (BWAPI) allows developers to retrieve information about the game state and interact with units in StarCraft, one of the most popular commercial RTS games [17]. Stratagus provides a free, cross-platform RTS engine that has been used directly in several research projects [18]. WARGUS uses Stratagus as the back-end for game play, but uses the entity data from the commercial game WarCraft II [19]. The same thing has been done with STARGUS, which uses the entity data from StarCraft [20]. ORTS, another RTS engine, provides a total programming environment for computational and artificial intelligence research, including a graphical client [21]. While other projects do not target a specific method, we designed Watercraft specifically for evolutionary computing approaches. In the next section, we describe how we configured Watercraft for our research as well as our search methods.

III. Methodology

We made several changes to the default Watercraft configuration. Since we evaluate the fitness of an individual by decoding the individual into a build order and simulating the result in Watercraft, we need to run Watercraft for every evaluation. With thousands of evaluations necessary, we turned off the graphics display and GUI during evaluation to simplify our simulations and reduce the amount of wall clock time a game takes to run. By disabling the OGRE graphics engine we greatly reduced the amount of processing time required to run a simulation. The ability to easily turn off the graphics makes WaterCraft especially suited for evolutionary computing approaches.

In this preliminary research we worked with only four types of units and the buildings needed for their production. Marines can be built quickly and cheaply, with barracks being the only infrastructure required beforehand, but with low offensive capabilities. Firebats are stronger than Marines, but require more infrastructure and cost more to produce. Vultures are the strongest unit, but cost the most and take the longest to produce. SCVs are used for gathering resources and building new structures, but have little offensive or defensive value. When building a structure, the SCV must move to the build location of the structure, and becomes unavailable until the structure is complete. When gathering resources, the SCV must move to the source of the resource, gather a small portion, and deliver the resource to a Command Center. Once the SCV delivers the resource to the Command Center, the resource is added to a bank that players use to pay for additional units and structures. When we initialize a game, each player starts with five SCVs and a Command Center. We place these units near sources of additional resources to decrease the amount of time spent gathering and delivering resources.

We created three hand-tuned "baseline" build-orders to compare against potential solutions generated by the GA and HC. We designed the first baseline to attack quickly with a small force. The baseline builds three additional SCVs for gathering minerals, followed by constructing five Marines, then attacking the player. This strategy challenges the player by quickly building enough units to attack before the player builds much. The second baseline does the same, but builds ten Marines instead of five. While much slower than the first strategy, the second baseline builds a much harder force to overcome if left uninterrupted. The third baseline builds two SCVs for gathering minerals, three SCVs for gathering gas, five Vultures, then attacks. This build-order also takes a long time to complete, but focuses on building fewer units that are individually stronger.

TABLE I
Unit Encodings

Bit Sequence	Action	Prerequisites
000-001	Build SCV (Gather Minerals)	None
010	Build Marine	Barracks
011-100	Build Firebat	Barracks, Refinery, Academy
101	Build Vulture	Barracks, Refinery, Factory
110	Build SCV (Gather Gas)	Refinery
111	Attack	N/A

Players typically determine how well they played by using the score at the end of a game. In the context of our search methods, the score also acts as a strategy's fitness. Our fitness calculation encourages players to find ways to destroy enemy units and structures, as shown in Equation 1. Where F_{ij} is the fitness of individual i against individual j, SR_i is the amount of resources spent by individual i, UD_j is the set of units owned by individual j that were destroyed, UC_k is the cost to build unit k, BD_j is the set of buildings owned by individual j that were destroyed, and BC_k is the cost to build building k.

$$F_{ij} = SR_i + 2 \sum_{k \in UD_j} UC_k + 3 \sum_{k \in BD_j} BC_k \qquad (1)$$

We represented a build order as a sequence of commands. Since GAs prefer a binary encoding, both the GA and the Hill-Climber worked with the same binary encoded sequence of commands - using GA terminology, this is our chromosome. We encoded the chromosome as a 15-length binary string. Every three bits of the binary string encodes a unit, as shown in Table I. When WaterCraft receives a chromosome, the game component that deals with the artificial player (Game AI or just AI), sequentially decodes the binary string and inserts a build action for each encoded unit into a queue. While decoding the binary string, the game AI verifies that all the prerequisites shown in Table I for the next encoded unit were previously inserted into the queue. If a prerequisite was not inserted into the queue already, the decoder will insert a build action for the prerequisite immediately before the encoded unit. Our game AI will attempt to issue build orders from the queue as quickly as possible. When the AI fail to execute the current action because of a lack of resources or pending prerequisite being built, the AI waits a short duration and reattempt to execute the action until the action succeeds.

A. *Exhaustive Search*

Exhaustive search evaluates all 2^{15} possible build-orders against all three of our baselines. We limited ourselves to 15-bit solutions since that was the maximum build-order length we could exhaustively search in a reasonable amount of time. Exhaustive search enables us to rank all possible 15-bit solutions, and compare the effectiveness of solutions found by GAs, HCs, as well as other search methods in the future.

B. Hill-climber

We use the bit-setting optimization hill-climber shown in Algorithm 1, which attempts to find an effective solution by sequentially flipping each bit and keeping the value with the highest fitness. We determine the fitness by playing a chromosome against all three baselines and taking the sum of the differences in scores, as shown in Equation 2. Where f_i is the fitness of chromosome i, j is a baseline in the set of all baselines B, F_{ij} is the fitness chromosome i received against baseline j, and F_{ji} is the fitness baseline j received against chromosome i.

Algorithm 1 Bit Setting Optimization Hill-climber

```
chromosome = initialize()
select first bit
evaluate(chromosome)
while not end of chromosome do
    flip current bit
    evaluate(chromosome)
    if fitness decreased then
        flip current bit back
    end if
    select next bit
end while
```

$$f_i = \sum_{j \in B} F_{ij} - F_{ji} \tag{2}$$

Hill-climber performance depends on the initial seed. We initialize this hill-climber with thirty-two different seeds: a chromosome set to all 0's, a chromosome set to all 1's, and thirty randomly generated chromosomes. Using these thirty-two different seeds enables us to obtain and report on statistically significant results.

C. Genetic Algorithm

When evaluating an individual for genetic operations such as selection, we used a teach set, scaled fitness, and fitness sharing as described by Rosin and Belew [24]. For our GA we modified the concept of a teach set; instead of creating a new teach set each generation, we populate the teach set solely with our baseline strategies. Usually, only chromosomes that can defeat many teach set members have a high fitness. However, shared fitness also gives chromosomes that only defeat a few teach set members few other chromosomes defeat, a high fitness. Chromosomes that defeat teach set members few others can, probably contain new and important innovations for winning. Giving these unique chromosomes a higher fitness prevents diverse niches from going extinct too quickly. We also multiply the shared fitness by the score the chromosome received against each defeated opponent. This allows us to identify chromosomes that not only win, but perform significantly better in the game. We calculated fitness sharing as shown in Equation 3. Where f_i^{shared} is the shared fitness of chromosome i, D_i is the set of teach set members chromosome i defeated, j a teach set member in D_i, j_l is the number of times j lost against all chromosomes, and F_{ij} is the fitness of chromosome i against teach set member j.

$$f_i^{shared} = \sum_{j \in D_i} \frac{1}{j_l} F_{ij} \tag{3}$$

j_l is the total number of individuals that baseline j lost to in the current population. We calculate j_l using Equation 4, where P is the set of all chromosomes in a population, and i is an element of P. $GameResult$ is a function that returns 1 if baseline j lost against individual i, and returns 0 if baseline j won against individual i, as show by Equation 5. Since Equation 3 only takes the sum of baselines that were defeated j_l can never be 0, since if a baseline j was never defeated j would not be in set D_i.

$$j_l = \sum_{i \in P} GameResult(j, i) \tag{4}$$

$$GameResult(j, i) = \left\{ \begin{array}{ll} 0, & F_{ji} >= F_{ij} \\ 1, & F_{ji} < F_{ij} \end{array} \right\} \tag{5}$$

When all population members have a shared fitness, we use linear fitness scaling with a factor of 1.5. At times there may be a population member who's shared fitness dwarfs all the other chromosome shared fitnesss, or where all population members have shared fitnesss that are very close. Fitness scaling helps even out selection pressure, making the children produced more diverse.

For the basic GA operations and parameters, we used uniform crossover with a 95% chance of crossover occurring and bit-mutation with a .1% chance of each individual bit changing value. The chromosomes are selected for crossover by using standard roulette wheel selection. Once we have created all the children chromosomes, we evaluate them against the same teach set, then use CHC selection to select chromosomes for the next population [25]. This prevents valuable information from being lost if our GA produces many unfit children.

IV. Results

Running 400 nodes in parallel allowed us to complete the exhaustive search in 16 hours. The vast majority of the available strategies end up losing to all three baselines as shown in Figure 3. This may not be very surprising since the baselines have the advantage of performing more than the five actions encodable in our chromosome representation. Despite this advantage, a substantial number of solutions find and exploit a weakness in at least one of the baselines, as shown in Figure 4. There are a few (difficult to see in the Figure) that beat two baselines but there are none that beat all three. Breaking down these results further, we can see from the Figure, the difference in difficulty that each the baseline provides against all possible builds.

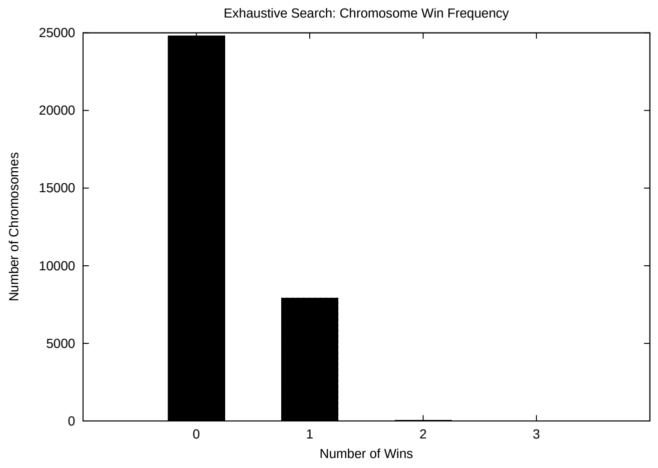

Fig. 3. The number of times exhaustive search finds counter-strategies to our baselines.

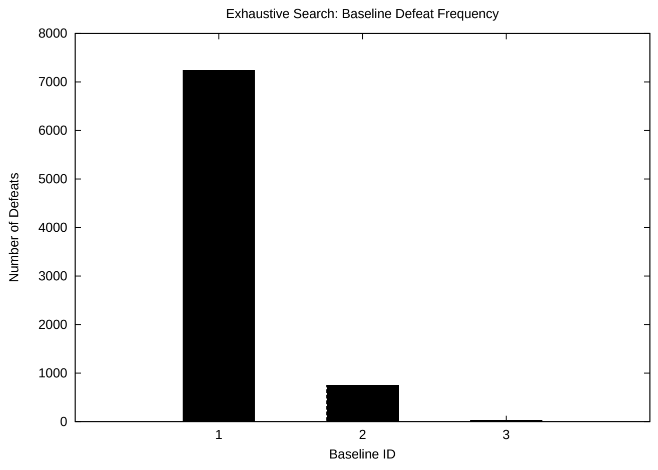

Fig. 4. The number of times the baselines lose in exhaustive search.

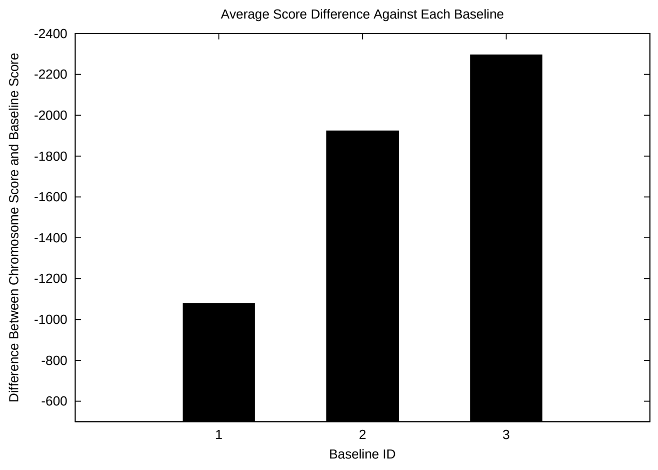

Fig. 5. The average difference in score between chromosomes and each baseline.

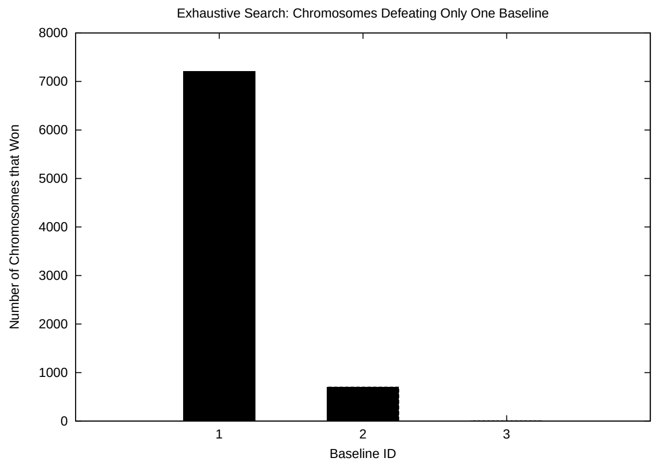

Fig. 6. The number of counter-strategies that defeat only one baseline, and the baselines they defeat.

Baseline 1 provides the easiest strategy to overcome, baseline 2 is harder, and baseline 3 rarely loses. The average scores also reflect these difficulties, as shown by Figure 5.

Most of the baseline losses can be attributed to strategies that are tuned solely for beating individual baselines, as shown by Figure 6. Though rare, the exhaustive search clearly shows that there exist strategies within our space of possible strategies, that can also beat two opponents, shown in Figure 7. There are no 15-bit strategies that beat all three baselines. Breaking these results down further, we see once again that baseline 3 rarely loses. As we can see from these exhaustive results, this problem provides a search space where solutions that beat multiple baselines are few and may be difficult to find.

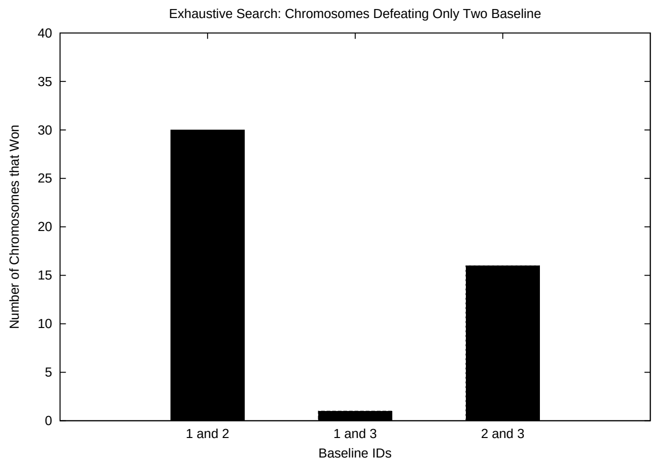

Fig. 7. The number of counter-strategies that defeat only two baselines, and the baselines they defeat.

Our hill-climber took 3 hours to run 32 different times with different seeds in parallel on 96 nodes. Out of thirty-two different seeds, only two seeds lead to an optimal solution that could beat both baseline 2 and baseline 3. Fifteen more seeds were able to find solutions within the top 20 solutions for defeating only baseline 1. The remaining seeds lead to solutions that lost against all baselines. However, the overall average of the scores was still better (less negative) than exhaustive search, as shown in Figure 8.

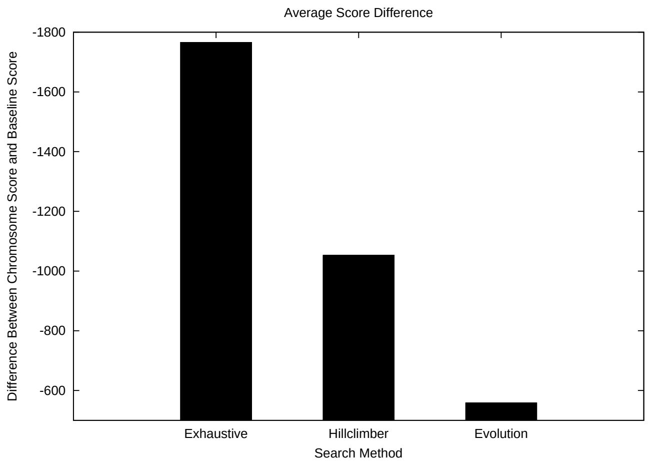

Fig. 8. The average difference in score from baseline scores for each search method.

The GA used a population of 50 for a maximum of 100 generations. Since one run of the GA took a total of 20 hours to run in parallel on 150 nodes, we ran the GA a total of only ten times with different random seeds. However the GA typically found the best solutions after only 6 hours (20 generations). Our results show that by playing against all three baselines, the GA converges to three unique optimal solutions. Two of the unique solutions defeat a pair of baselines; baselines 1 and 2, and the baselines 2 and3. These strategies are also the highest scoring strategies out of all strategies that defeat the same two baselines. The third solution found was the optimal strategy that defeated only baseline 2. Because of our fitness sharing, the best chromosome of each generation would cycle between three strategies. As strategies that could defeat baseline 1 and baseline 3 start to take over the population, strategies that can defeat baseline 2 start to die out but are given more weight. Eventually the shared fitness crosses a threshold where strategies that only defeat baseline 2 are given so much weight they briefly have the highest shared fitness. As strategies that defeat baseline 2 start to make a comeback in the population, the weight given to those strategies becomes lower. Eventually the chromosomes with the highest shared fitness becomes strategies that can defeat baseline 1 and baseline 2 (although the strategy scores less than a strategy that defeats only baseline 2). Then as these strategies start to take over the population, the cycle restarts. By generation 100, the population consists entirely of these three strategies.

V. Conclusion and Future Work

This paper compared genetic algorithms with bit-setting hill-climbing to generate strategies for beating opponents in RTS games. We used three very different hand-coded strategies as baselines upon which to make our comparisons. Finally, we restricted our space of strategies to 2^{15} in order to be able to exhaustively evaluate all 2^{15} strategies against our three baselines and thus enable absolute comparison of genetic algorithm performance versus a much faster deterministic bit-setting hill-climber.

Results show that the hill-climber quickly finds good counter-strategies (to our baselines), but is not very good at finding the best counter-strategies robustly. It finds the best only about 6% of the time starting with different random seeds. The genetic algorithm, on the other hand, robustly (100% of the time) finds the best possible strategies but can take much longer to find them.

These results help specify trade-offs to be made when choosing between GA and HCs in the kind of strategy spaces found in RTS games. The results also agree well with our understanding of genetic algorithm and hill-climbing theory. Our results indicate that if you are interested in a quick satisficing solution but not overly worried about optimality, a hill-climber will probably work best. On the other hand, if you are interested in high quality counter-strategies you should probably use a genetic algorithm.

We are interested in quickly finding high quality adaptive counter-strategies so that we can design human competitive RTS game players. This paper's results indicate that speeding up the genetic algorithm, perhaps through case-injection, or other method may help and will be the subject of our future work. In addition, although the GA robustly finds high quality counter-strategies, these strategies themselves may only work well against the three baselines that they were evaluated against. That is, the strategies found by the GA may themselves not be robust and able to beat many of the other $2^{15} - 3$ strategies in our strategy space. We are thus also considering co-evolutionary approaches to strategy generation and in future work plan on combining case-injected genetic algorithms with co-evolution to make progress in designing highly competent RTS game players. This will have significant applications for generating competent opposing forces in military training simulations and wargaming.

Acknowledgment

This research is supported by ONR grant N00014-09-1-1121.

References

[1] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of the 7th international conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*, ser. ICCBR '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 164–178. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74141-1_12
[2] S. Louis and J. McDonnell, "Learning with case-injected genetic algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 4, pp. 316 – 328, aug. 2004.
[3] S. Louis and C. Miles, "Playing to learn: case-injected genetic algorithms for learning to play computer games," *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 669 – 681, dec. 2005.

[4] P. Avery and S. Louis, “Coevolving influence maps for spatial team tactics in a rts game,” in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, ser. GECCO ’10. New York, NY, USA: ACM, 2010, pp. 783–790. [Online]. Available: http://doi.acm.org/10.1145/1830483.1830621

[5] M. T. Tsapanos, K. C. Chatzidimitriou, and P. A. Mitkas, “A zeroth-level classifier system for real time strategy games,” in *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 02*, ser. WI-IAT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 244–247. [Online]. Available: http://dx.doi.org/10.1109/WI-IAT.2011.177

[6] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, “Adaptive game ai with dynamic scripting,” *Mach. Learn.*, vol. 63, no. 3, pp. 217–248, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10994-006-6205-6

[7] M. Ponsen, H. Munoz-Avila, P. Spronck, and D. W. Aha, “Automatically generating game tactics through evolutionary learning,” *AI Magazine*, vol. 27, no. 3, pp. 75–84, 2006. [Online]. Available: http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1894

[8] B. G. Weber and M. Mateas, “Case-based reasoning for build order in real-time strategy games,” in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2009)*, 10/2009 2009.

[9] D. Churchill and M. Buro, “Build order optimization in starcraft,” in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2011)*, 10/2011 2011.

[10] C. K. Tong, C. K. On, J. Teo, and A. M. J. Kiring, “Evolving neural controllers using ga for warcraft 3-real time strategy game,” in *Proceedings of the 2011 Sixth International Conference on Bio-Inspired Computing: Theories and Applications*, ser. BIC-TA ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 15–20. [Online]. Available: http://dx.doi.org/10.1109/BIC-TA.2011.70

[11] T. W. Sandberg, “Evolutionary multi-agent potential field based ai approach for ssc scenarios in rts games,” Master’s thesis, University of Copenhagen, February 2011.

[12] J. McCoy and M. Mateas, “An integrated agent for playing real-time strategy games,” in *Proceedings of the 23rd national conference on Artificial intelligence - Volume 3*, ser. AAAI’08. AAAI Press, 2008, pp. 1313–1318. [Online]. Available: http://dl.acm.org/citation.cfm?id=1620270.1620278

[13] F. Safadi, R. Fonteneau, and D. Ernst, “Artificial intelligence design for real-time strategy games,” in *NIPS Workshop on Decision Making with Multiple Imperfect Decision Makers*, 2011.

[14] B. G. Weber, M. Mateas, and A. Jhala, “Building human-level ai for real-time strategy games,” in *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, AAAI Press. San Francisco, California: AAAI Press, 2011.

[15] D. Livingstone, “Coevolution in hierarchical ai for strategy games,” in *CIG’05*, 2005.

[16] S. Wintermute, J. Xu, and J. Laird, “Sorts: A human-level approach to real-time strategy ai,” *Ann Arbor*, vol. 1001, pp. 48 109–2121, 2007.

[17] “Bwapi: An api for interacting with starcraft: Broodwar.” [Online]. Available: http://code.google.com/p/bwapi/

[18] M. J. V. Ponsen, S. Lee-urban, H. Muoz-avila, D. W. Aha, and M. Molineaux, “Stratagus: An open-source game engine for research in real-time strategy games,” Naval Research Laboratory, Navy Center for, Tech. Rep., 2005.

[19] T. W. Team, “Wargus,” 2011. [Online]. Available: http://wargus.sourceforge.net

[20] “Stargus,” 2009. [Online]. Available: http://stargus.sourceforge.net/

[21] M. Buro, “Orts - a free software rts game engine,” 2005. [Online]. Available: http://skatgame.net/mburo/orts/

[22] T. K. S. Ltd, “Ogre open source 3d graphics engine,” February 2005. [Online]. Available: http://www.ogre3d.org/

[23] B. Entertainment, “Starcraft,” March 1998. [Online]. Available: http://us.blizzard.com/en-us/games/sc/

[24] C. D. Rosin and R. K. Belew, “New methods for competitive coevolution,” *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, Mar. 1997. [Online]. Available: http://dx.doi.org/10.1162/evco.1997.5.1.1

[25] L. J. Eshelman, “The chc adaptive search algorithm : How to have safe search when engaging in nontraditional genetic recombination,” *Foundations of Genetic Algorithms*, pp. 265–283, 1991. [Online]. Available: http://ci.nii.ac.jp/naid/10000024547/en/