Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

Fast Random Genetic Search for Large-Scale RTS Combat Scenarios

Corey Clark, Anthony Fleshner
Guildhall
Southern Methodist University
Plano, TX, United States

Abstract

This paper makes a contribution to the advancement of artificial intelligence in the context of multi-agent planning for large-scale combat scenarios in RTS games. This paper introduces Fast Random Genetic Search (FRGS), a genetic algorithm which is characterized by a small active population, a crossover technique which produces only one child, dynamic mutation rates, elitism, and restrictions on revisiting solutions. This paper demonstrates the effectiveness of FRGS against a static AI and a dynamic AI using the Portfolio Greedy Search (PGS) algorithm. In the context of the popular Real-Time Strategy (RTS) game, StarCraft, this paper shows the advantages of FRGS in combat scenarios up to the maximum size of 200 vs. 200 units under a 40 ms time constraint.

Introduction and Background

Multi-agent planning is a part of RTS artificial intelligence. This concept refers to the collective planning of many individual units (a single entity in a video game. For example: a single 'Marine' unit in StarCraft) belonging to a single AI Commander (the overarching artificial intelligence system that governs the behavior of a non-player character (NPC) in a video game).

In the context of RTS games, multi-agent planning usually involves all units, vehicles, structures, resources, etc. owned by one player. An AI Commander controlling an NPC in an RTS game regulates the actions of every unit at every step of the game. Considering only units, a player character or NPC in StarCraft can have up to 200 units to command at any given time. Furthermore, each unit can have a widely varying degree of actions it can perform at any time including movement, building, attacking, gathering, pathfinding, etc.

This paper addresses the problem of multi-agent planning for combat scenarios in RTS games, scaling up to the largest possible size of 200 vs. 200 units. A combat scenario consists of two armies of equal initial size starting in separated and symmetrical states. Each unit in the army (the collection of all units under the control of an AI Commander) can move or attack an enemy unit. When a unit is attacked, the attacked unit loses health points until its total health points reaches zero. When a unit reaches zero health points, it is considered "dead" and removed from the scenario. An AI Commander "wins" a scenario when all of the enemy's units are removed from the combat scenario.

The retail version of StarCraft runs at 24 frames per second, which means each frame refreshes after 41.67 ms. In pursuit of a real-time solution, this paper explores an algorithm capable of improving the combat state of a StarCraft AI Commander in 40 ms or less.

Related Work

Churchill, D. et Buro, M. 2013, implemented Portfolio Greedy Search (PGS), a hill climbing greedy search that creates a portfolio (a set of strategies available to the AI Commander) of move possibilities and compared to the following techniques:

- Alpha-beta – A search technique that involves removing branches (pruning) of the search tree that do not influence the final decision.
- UCT – A Monte Carlo Tree Search (MCTS) technique that involves a balance between selecting the best move found so far and exploring new moves.
- UCT Considering Durations (UCTCD) – An implementation of UCT search which considers simultaneous moves with durative actions, instead of alternating moves with identical durations.

PGS was shown to outperform Alpha-Beta, UCT, and UCTCD for army sizes greater than 32 with a separated initial state. PGS achieved >90%-win percentage over Alpha-beta and a 90%-win percentage over UCT for army sizes 32 and 50, when initialized with a separated state (Churchill and Buro 2013).

Wang et al. 2016 developed Portfolio Online Evolution (POE), which combined PGS with Online Evolution (Justesen, Mahlmann, and Togelius 2016). POE utilizes evolution with a static mutation rate rather than tree search to identify a sequence of moves for a given portfolio. POE outperformed PGS with an average win rate around 75%, but performed slightly better with smaller army sizes. POE outperformed UCT and UCTCD in large to moderate army sizes and had near a 50% win rate for small size armies (Wang et al. 2016).

Static AI

A static AI Commander controls individual units according to a single strategy (a set of rules that governs the behavior of a unit). Any strategy may be unique or a combination of other strategies, but the overall behavior of AI units is predictable and prone to exploitation (Churchill, Saffidine, and Buro 2012). This research utilizes eight different strategies from the SparCraft combat simulator (Churchill 2012) including:

- AttackClosest: Attack the closest enemy in range. If no enemies are in range, move toward the closest enemy.
- AttackWeakest: Attack the enemy in range with the lowest health. If no enemies are in range, move toward the closest enemy.
- AttackDPS: Attack the enemy in range with the highest DPS (damage per second) / health value. If no enemies are in range, move toward the closest enemy.
- NOKDPS: Attack the enemy in range with the highest DPS / health value that has not been dealt lethal damage. If no enemies are in range, move toward the closest enemy. This behavior prevents other friendly units from attacking enemy units that would die anyway, effectively preventing "overkill".
- Kiter: If this unit can currently attack, attack the closest enemy. If this unit cannot currently attack and it is in range of an enemy, move away. If this unit can attack and no enemies are in range, move toward the closest enemy. This strategy attempts to move the unit away from enemies while it cannot attack, to prevent enemies from attacking it, effectively "kiting" the enemy.
- KiterDPS: Follows the same behavior as Kiter, except KiterDPS targets the enemy in range with the highest DPS / health value.
- KiterNOKDPS: Follows the same behavior as KiterDPS, except KiterNOKDPS prevents overkill from other friendly units. This strategy is effectively Kiter combined with NOKDPS.
- Cluster: This strategy moves all units towards a central position.

A static AI Commander commands all units to follow the same strategy. For example, an AI Commander following the 'AttackClosest' strategy will command each unit to attack the closest enemy in range. The units belonging to the AI Commander follow the same strategy throughout an entire battle. A static AI does not have the adaptability of a dynamic AI. In a dynamic system, the AI Commander can change the strategy of its units as the combat scenario evolves.

Dynamic AI

Evaluation Function

The evaluation function determines the score of a current state of all units owned by the AI Commander. This evaluation function returns a single score as a numerical representation of the value of any given state. A higher score than the enemy score represents and advantageous state, while a lower score than the enemy represents a disadvantageous one. The evaluation function for RTS games used by A. Kovarsky et M. Buro (Kovarsky and Buro 2005) is the Life-Time-Damage-2 (LTD2) function. The LTD2 function was also used by Churchill, D. et Buro, M. (Churchill and Buro 2013) in experiments comparing PGS to Alpha-Beta, UCT, and UCTCD. LTD2 is an evaluation function which favors having many units over a single unit (all summed health equal). LTD2 also favors keeping alive units which deal the highest damage (damage per frame) (Erickson and Buro 2014).

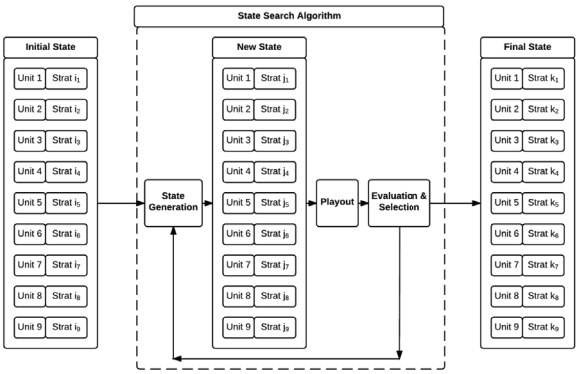

Figure 1: State search diagram for PGS and FRGS.

$$LTD2(s) = \Sigma_{\mu \in U1} \sqrt{hp(u)} * dpf(u) - \Sigma_{\mu \in U2} \sqrt{hp(u)} * dpf(u) \quad (1)$$

Where s, $\mu \in U1$, $\mu \in U2$, $hp(u)$, and $dpf(u)$ represent state, every unit in the AI Commander's army, every unit in the simulated enemy army, health points of a unit, and damage per frame of a unit, respectively. The LTD2 evaluation function calculates a health-damage number for each unit. For each unit, the square root of the unit's health is multiplied by its damage per frame value to find its health-damage number. Each unit's health-damage number is added together to find a single summation of all health-damage numbers for the AI Commander. A state score for the AI Commander is given by its summation score subtracted from the summation score of the enemy.

Playouts

A playout is defined as a simulation from the current state until a game-ending state is achieved. The LTD2 score of the game-ending state represents the outcome of the playout. If a playout returns a higher LTD2 score than the current state, then the playout state is better than the current state. In order to reduce the total time spent running playouts, the number of simulated turns is limited to 100 during each playout to match playout limitation in Churchill, D. et Buro, M. (Churchill and Buro 2013).

Figure 1 represents the structure PGS and FRGS. Both algorithms begin with an initial state that is seeded or randomized. Each algorithm has a unique state generation method. At the end of each iteration, the state is evaluated after a 100 turn playout. When an evaluated state scores higher than the

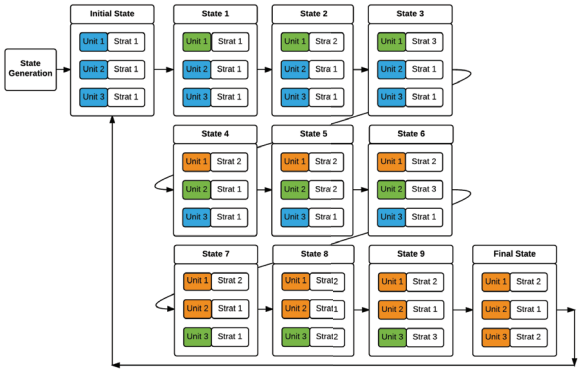

Figure 2: State generation process for PGS. Blue represents default strategy for unit, green is unit under test during evaluation state and orange represents units strategy has been selected.

current best state, the current best state is updated. When the 40 ms time limit is reached, the current best state is returned as the final state.

Portfolio Greedy Search

PGS is a hill climbing algorithm, presented in Churchill, D. et Buro, M. (Churchill and Buro 2013), which considers each strategy for each combat unit and chooses the best one based on its LTD2 score.

1. First, PGS calculates an initial seed state for both the AI Commander and a simulated enemy Commander by selecting a strategy from the portfolio [Algorithm 1, Lines 7-8, 19-24]. In this implementation of PGS, the initial seed strategy is chosen randomly from the portfolio and applied to all units belonging to the AI Commander. The initial seed is chosen randomly to reduce the amount of time spent calculating an initial seed. The calculation of an initial seed used in Churchill, D. et Buro, M. (Churchill and Buro 2013) requires M playouts and evaluations, where M is the number of strategies in the portfolio.
2. After the initial seed is chosen, the algorithm enters the improve cycle [Algorithm 1, Lines 10, 26-47]. During each iteration of improvement, a single unit strategy is changed and an LTD2 evaluation determines the score of the new state. If the new state score is higher than the previous state score, the unit's new strategy improved the state. Then, the execution considers the next strategy in the portfolio. After all strategies are evaluated for a single unit, the next unit is evaluated in the same way. The algorithm then repeats the improvement process for I iterations.
3. When the time limit (40 ms) is reached, the current evaluation finishes and the algorithm returns the best state calculated.

In Figure 2, the AI Commander has an army size of 3 units and a portfolio of 3 strategies. States are generated by altering the strategy of a single unit. The algorithm creates N*M states per iteration, where N is the number of units and M is

Algorithm 1 Portfolio Greedy Search

```
1:  Portfolio P                                          ▷ Script Portfolio
2:  Integer I                                            ▷ Improvement Iterations
3:  Integer R                                            ▷ Improvement Responses
4:  Script D                                             ▷ Default Script
5:
6:  procedure PGS(State s, Player p)
7:      enemy[s.numUnits(opponent(p)].fill(D)
8:      self[] ← GetRandomSeedPlayer(s, opponent(p),
9:                                   self)
10:     self ← Improve(s, p, self, enemy)
11:     for r=1 to R do
12:         enemy ← Improve(s, opponent(p), enemy,
13:                         self)
14:         self ← Improve(s, p, self, enemy)
15:     end for
16:     return generateMoves(self)
17: end procedure
18:
19: procedure GetRandomSeedPlayer(State s, Player p, Script e[])
20:     Script self[s.numUnits(p)]
21:     randomScript ← rand(0, p.numScripts)
22:     self.fill(randomScript)
23:     return self
24: end procedure
25:
26: procedure Improve(State s, Player p, Script self[], Script e[])
27:     Script self[s.numUnits(p)]
28:     for i=1 to I do
29:         for u=1 to self.length do
30:             if timeElapsed > timeLimit then
31:                 return
32:             end if
33:             bestValue ← −∞
34:             bestScript ← null
35:             for Script c in P do
36:                 self[u] ← c
37:                 value ← Playout(s, p, self, e)
38:                 if value > bestValue then
39:                     bestValue ← value
40:                     bestScript ← c
41:                 end if
42:                 self[u] ← bestScript
43:             end for
44:         end for
45:     end for
46:     return self
47: end procedure
```

the number of strategies in the portfolio. After M strategies have been created for a single unit, the AI Commander picks the highest scoring (LTD2) state and assigns the associated strategy to the unit. The AI Commander continues in this way until all units are evaluated. A state selection occurs N times per iteration, effectively picking the current best strategy for each unit. The final state of each iteration is used as the initial state of the next iteration.

Fast Random Genetic Search

Fast Random Genetic Search (FRGS) is characterized by a small active population, a crossover technique which produces only one child, dynamic mutation rates, elitism, and

Algorithm 2 Fast Random Genetic Search

```
1: Portfolio P                                              ▷ Script Portfolio
2: Integer R                                                ▷ Improvement Responses
3: Script D                                                 ▷ Default Script
4: procedure FRGS(State s, Player p)
5:     enemy[s.numUnits(opponent(p))].fill(D)
6:     self[] ← GetRandomSeedPlayer(s, opponent(p),
7:                                  self)
8:     self ← Improve(s, p, self, enemy)
9:     for r=1 to R do
10:        enemy ← Improve(s, opponent(p), enemy,
11:                        self)
12:        self ← Improve(s, p, self, enemy)
13:    end for
14:    return generateMoves(self)
15: end procedure
16: procedure GetRandomSeedPlayer(State s, Player p, Script e[])
17:    Script self[s.numUnits(p)]
18:    randomScript ← rand(0, p.numScripts)
19:    self.fill(randomScript)
20:    return self
21: end procedure
22: procedure Improve(State s, Player p, Script self[], Script e[], Int evalsPerTier)
23:    Map(Score, Script[s.numUnits(p)][]) population
24:    score ← Playouts(s, p, self, e)
25:    population[score].add(self)
26:    while timeElapsed < timeLimit do
27:        Script chromo[] ← self
28:        currentMutateTier ←
29:                    (evalCount/evalsPerTier) + 1
30:        numToMutate ←
31:                    Max(1, numUnits/currentMutateTier)
32:        if population.numScoreValues > 1 then
33:            chromo ←
34:              CreateChild(population[0][0], population[1][0])
35:        end if
36:        chromo ← Mutate(chromo, numToMutate)
37:        score ← Playout(s, p, chromo, e)
38:        population[score].add(chromo)
39:    end while
40:    self ← population[0][0]
41:    return self
42: end procedure
43: procedure CreateChild(Script chromo1[], Script chromo2[])
44:    Script child[]
45:    child.randomHalfUnits ←
46:                    chromo1.randomHalfUnits
47:    child.otherHalfUnits ←
48:                    chromo2.otherHalfUnits
49:    return child
50: end procedure
51: procedure Mutate(Script chromo, int numUnitsToMutate)
52:    Script mutatedChromo
53:    mutatedChromo ← chromo
54:    for u=0 to numUnitsToMutate do
55:        mutatedChromo[nextRandUnit] ←
56:                    randomScript
57:    end for
58:    return mutatedChromo
59: end procedure
```

restrictions on revisiting solutions. Given the need to complete the heuristic search within a small fraction of a second, the effective chromosome population size is limited in the present case to only two solutions. As such, the crossover rate is set to 1. The two solutions in the population are bred at each step. Since solution evaluation is costly in terms of CPU time, the crossover operation only produces one new solution per evaluation. Dynamic mutation rates have shown to increase convergence (Jun Zhang, Chung, and Hu 2004; Bck 1992; Fogarty 1989; Hesser and Mnner 1991) and are used to increase exploration early on and increase solution refinement in the later iterations. During the initial generations, a large mutation rate is used to create a genetically guided random search. In the later generations, the child solution is composed primarily of the genes of the parents. The child solution is retained in the population only if its fitness value is superior to at least one parent. If so, the child solution replaces the lowest quality parent solution. This is effectively the same as an elitist strategy where the number of elite solutions equals the population size. While having such a high rate of elitist solutions usually limits the population diversity, this is balanced by a high, but dynamic mutation rate. Finally, the genetic search maintains a memory of evaluated solutions and verifies the child is a new solution, otherwise mutation is again applied.

1. First, FRGS calculates an initial seed state for both the AI Commander and a simulated enemy by selecting a strategy from the portfolio. The initial seed is chosen randomly from the portfolio and applied to all units belonging to the AI Commander. An initial evaluation is computed and a <score, state> pair (chromosome) is added to the population of potential solutions [Algorithm 2, Lines 6-8, 16-21].

2. After the initial seed is chosen and evaluated, the algorithm enters the improve cycle. During each iteration of improvement, a new state is generated by combining the top two states in the population [Algorithm 2, Lines 8, 22-42]. The crossover method forms a new state from a random 50% of the first state and the complementary 50% from the second state [Algorithm 2, Lines 33, 43-50].

3. After the crossover, a random mutation is applied to a variable number of units [Algorithm 2, Lines 36, 51-59]. The number of units mutated is inversely correlated with the number of iterations. The first iteration has the highest mutation chance, up to 100% of the units are mutated. The number of mutated units decreases as the number of iterations increases. The number of units chosen for mutation is the number of units divided by the current mutation tier. The current mutation tier is calculated by dividing the number of evaluations completed by the evalsPerTier number. The evalsPerTier number is an arbitrary number which was set at '10' for this paper [Algorithm 2, Lines 28-30, 36]. If the mutated state is identical to a chromosome already in the population, the mutation process is applied again until a unique solution is created. This process promotes a high variation chance in the initial generation of the population and a low variation chance in the

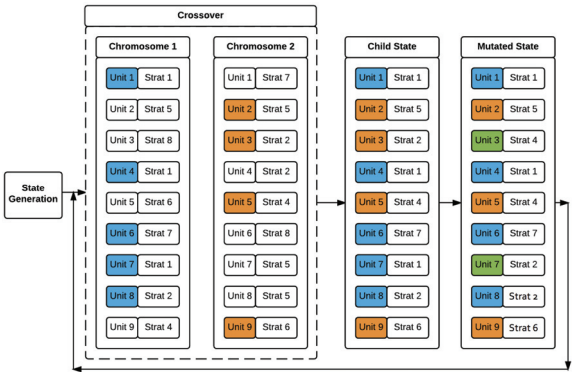

Figure 3: State generation process for FRGS. Blue and orange indicates strategies selected from Chromosome 1 and 2 respectively for child state, while green represents randomly mutated strategies of child

later iterations. The mutation process also prevents duplicate entries in the population.

4. When the time limit (40 ms) is reached, the current state finishes evaluation and the algorithm returns the best state calculated.

Figure 3 shows the state generation method of FRGS. In Figure 3, chromosome 1 and chromosome 2 represent the two highest LTD2 scoring states in the population. The two states are combined by selecting a random half of the first state and the complementary half from the second state. Then, a variable number of units' strategies are randomly changed during the mutation step.

Explanation of Experiments

Hardware / Software

All experiments were performed on an Intel Core i7-4810MQ CPU @2.80GHz. Experiments ran in a single thread only. Experiments were programmed in C++ using Visual Studio 2013.

SparCraft Scenarios and Units

Each scenario simulated in SparCraft consisted of two AI Commander with a varying number of units. In this 1v1 (1 Commander vs. 1 Commander) scenario, each Commander controlled the same number of units, ranging from 5 – 200 (maximum allowed in StarCraft). A total of 100 battles were simulated for each army size of 5, 10, 20, 40, 60, 80, 100, 120, 140, 160, and 200. For all experiments, each army consisted of entirely 'Marine' units from the 'Terran' faction of StarCraft. The Marine unit is a basic ranged combat unit.

Each unit was generated symmetrically about a midpoint in the battle arena, then translated some distance towards opposing sides of the arena (Churchill 2012). The starting positions of both armies were symmetrical about the midpoint of the arena, but separated by a random distance. The separated starting positions closely simulated the meeting of two opposing StarCraft armies in an actual game.

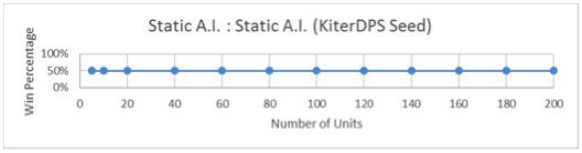

Figure 4: Results of KiterDPS strategy vs. KiterDPS strategy.

The SparCraft arena consisted of an open map with a fixed boundary. All units could move freely while remaining within the bounds of the arena. The SparCraft simulator restricted unit movement to four directions (forward, backward, left, right) to reduce the number of possible unit actions (Churchill and Buro 2013).

Search Parameters

The following parameters were used for both search techniques:

- Time limit: 40 ms
- Improvement Iterations: No limit
- Response Iterations: 0
- Portfolio: AttackClosest, AttackWeakest, AttackDPS, NOKDPS, Kiter, KiterDPS, KiterNOKDPS, Cluster

The search algorithms were not limited by a pre-defined number of iterations. The only restriction on improvement was the 40 ms time limit. For both search techniques, there was no time spent calculating an initial seed. The seed was chosen for both Commanders before-hand or selected randomly to ensure that no extra time was spent initially seeding either algorithm. Experiments were run for an arbitrary starting seed (KiterDPS) and a random seed. In these experiments, the number of responses was set to 0. A response involves running the improvement algorithm on the simulated enemy and subsequently improves the final solution. A response value of 0 was chosen to show the baseline performance of both algorithms and was also used in Churchill, D. et Buro, M. (Churchill and Buro 2013).

Results

Figure 4 is included to demonstrate the validity of the underlying system. In this test, both armies simply chose the KiterDPS strategy at every stage of execution. Since both armies were seeded with separated and symmetrical states, the outcome (win percentage) at all army sizes was 50%. With all factors equal, a fair system is expected to yield a 50%-win percentage for both armies. These results verify the validity and integrity of the combat simulation system.

Additional test were executed between PGS and FRGS against the KiterDPS strategy. In both experiments, the dynamic AIs were seeded with KiterDPS. PGS and FRGS maintained a 100%-win ratio for all army sizes against the static AI. This result was expected as both dynamic AI are seeded with KiterDPS and any improvement made by PGS or FRGS produced a win.

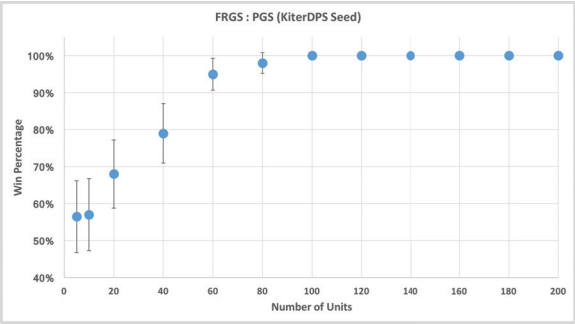

Figure 5: Results of FRGS vs. PGS with KiterDPS seed for both algorithms. Error bars show 95% confidence interval.

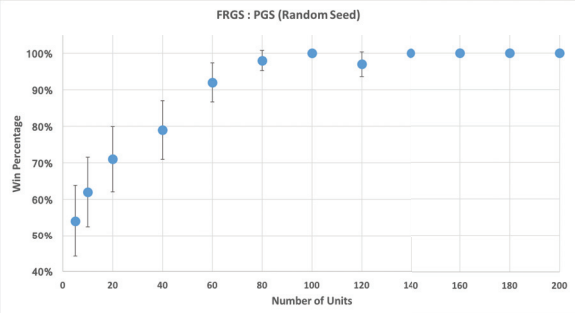

Figure 6: Results of FRGS vs. PGS with random seed for both algorithms. Error bars show 95% confidence interval.

FRGS vs. PGS

Figure 5 and Figure 6 show results of the FRGS against PGS when seeded with a single strategy (KiterDPS) and with a random seed for army sizes ranging from 5 – 200. In both instances, FRGS outperformed PGS in terms of win percentage. The win percentage of FRGS vs. PGS is correlated logarithmically to the size of the armies.

As shown in the above graphs, Figure 5 and Figure 6, FRGS achieved a 100%-win percentage over PGS at maximum and near-maximum size armies. Even at small size armies down to 5 units, FRGS won more than 50% of the time. As the number of strategies in the portfolio and number of units increased, FRGS further outperformed PGS.

The following graphs, Figure 7 and Figure 8, show the relationship between score and time for both algorithms. Each graph shown is a single frame of a single battle. Every point on the graph represents time at which a new state is selected (x-axis) and the score of that state (y-axis) for that frame.

In Figure 7 & 8, FRGS completed more state selections than PGS. This does not mean that FRGS completed more evaluations. PGS completed roughly the same number of evaluations, but PGS' evaluations were spent analyzing single changes to single units. In the experiment with 20 units, PGS was able to analyze 5 units (25%) and in the experiment with 80 units, PGS was able to analyze 2 units (2.5%). Further analysis of the PGS algorithm shows the following attributes.

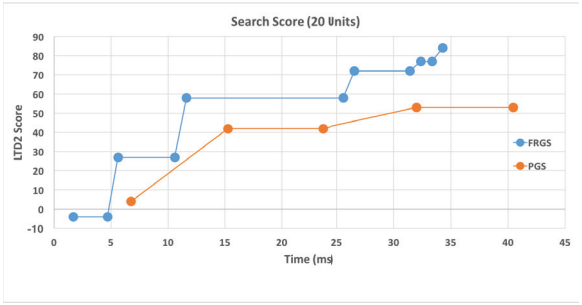

Figure 7: LTD2 score and time (ms) graph for FRGS and PGS with an army size of 20 units over a single frame (40 ms).

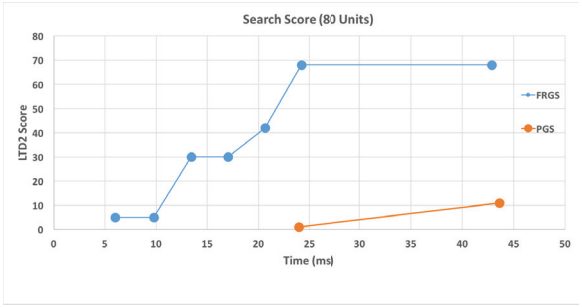

Figure 8: LTD2 score and time (ms) graph for FRGS and PGS with an army size of 80 units over a single frame (40 ms).

Attributes of PGS A.)PGS evaluations reflect a change to a single unit. B.)Searches a small portion of the total search space with a large number of strategies (8) and a large number of units (up to 200). C.)N*M evaluations and M state selections for one full pass over all units where N is the number of units and M is the number of strategies. D.)When time runs out, PGS has only analyzed a small percentage of units out of a potential of 200.

Attributes of FRGS A.)FRGS changes the strategy for up to 100% of units and then evaluates the entire state based on that change. The number of units changed per evaluation decreases as more iterations are completed. B.)Portfolio size does not affect search time. C.)FRGS evaluations reflect entire army changes, instead of per unit changes. D.)When time runs out, FRGS has analyzed a diverse set of states and picked the current best state.

Conclusion and Future Work

This paper presented the FRGS algorithm for large-scale RTS combat scenarios. Experiments were designed and executed to test the performance of a PGS and FRGS with a large portfolio (8 strategies) and up to maximum size (200 units) armies allowed in StarCraft using 1000 simulations. Results have shown that FRGS outperforms PGS when armies start in separated states. FRGS achieves 100%-win percentage as low as 100 units when the starting strate-

gies are seeded randomly. For a large number (8) of strategies in the portfolio, FRGS achieves >50%-win percentage over PGS at an army size as low as 5 units.

While the results have shown that FRGS outperforms PGS under certain conditions, FRGS emphasizes early exploration. Due to early exploration, FRGS spends time exploration a wide variety of states in the early stages of execution. While one of the early states has a chance of outscoring the initial state, FRGS does not quickly refine any state. FRGS quickly finds a wide variety of states and slowly lowers its mutation rate and subsequently increases the refinement rate in the later iterations.

FRGS struggles to find a better solution when the initial state is seeded with a high-scoring state. Given its high early exploration, the chances of improving a high-scoring state are low. In the case of a high-scoring initial state, FRGS will likely explore much weaker states while making no improvement. Although, as the mutation rate decreases, FRGS will be able to refine the initial state.

A potential improvement to FRGS involves limiting the early exploration phase when a high-scoring (LTD2) state is found. After a high scoring is generated, ending the exploration phase allows FRGS to enter a refinement phase. This improvement will increase the performance of FRGS for smaller sized armies (<100 units).

Knowing the strengths and weaknesses of FRGS allows future works to incorporate a hybrid solution between FRGS, PGS, and other search algorithms.

References

Bck, T. 1992. Self-Adaptation in Genetic Algorithms. In *Proceedings of the First European Conference on Artificial Life*, 263–271. MIT Press.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in starcraft. 1–8. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *In Proceedings of the AIIDE Conference*, 112–117.

Churchill, D. 2012. SparCraft: Open Source StarCraft Combat Simulation.

Erickson, G. K. S., and Buro, M. 2014. Global State Evaluation in StarCraft. In *AIIDE*.

Fogarty, T. C. 1989. Varying the Probability of Mutation in the Genetic Algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, 104–109. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hesser, J., and Mnner, R. 1991. Towards an Optimal Mutation Probability for Genetic Algorithms. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, 23–32. London, UK, UK: Springer-Verlag.

Jun Zhang; Chung, H.; and Hu, B. 2004. Adaptive probabilities of crossover and mutation in genetic algorithms based on clustering technique. 2280–2287. IEEE.

Justesen, N.; Mahlmann, T.; and Togelius, J. 2016. *Online Evolution for Multi-action Adversarial Games*. Cham: Springer International Publishing. 590–603.

Kovarsky, A., and Buro, M. 2005. Heuristic Search Applied to Abstract Combat Games. In Hutchison, D.; Kanade, T.; Kittler, J.; Kleinberg, J. M.; Mattern, F.; Mitchell, J. C.; Naor, M.; Nierstrasz, O.; Pandu Rangan, C.; Steffen, B.; Sudan, M.; Terzopoulos, D.; Tygar, D.; Vardi, M. Y.; Weikum, G.; Kgl, B.; and Lapalme, G., eds., *Advances in Artificial Intelligence*, volume 3501. Berlin, Heidelberg: Springer Berlin Heidelberg. 66–78. DOI: 10.1007/11424918_9.

Wang, C.; Chen, P.; Li, Y.; Holmgrd, C.; and Togelius, J. 2016. Portfolio online evolution in starcraft.