Continual Online Evolutionary Planning for In-Game Build Order Adaptation in StarCraft

Niels Justesen
IT University of Copenhagen
noju@itu.dk

Sebastian Risi
IT University of Copenhagen
sebr@itu.dk

ABSTRACT

The real-time strategy game StarCraft has become an important benchmark for AI research as it poses a complex environment with numerous challenges. An important strategic aspect in this game is to decide what buildings and units to produce. StarCraft bots playing in AI competitions today are only able to switch between predefined strategies, which makes it hard to adapt to new situations. This paper introduces an evolutionary-based method to overcome this challenge, called *Continual Online Evolutionary Planning* (COEP), which is able to perform in-game adaptive build-order planning. COEP was added to an open source StarCraft bot called UAlbertaBot and is able to outperform the built-in bots in the game as well as being competitive against a number of scripted opening strategies. The COEP augmented bot can change its build order dynamically and quickly adapt to the opponent's strategy.

CCS CONCEPTS

•**Applied computing** → **Computer games;** *Multi-criterion optimization and decision-making;*

KEYWORDS

Evolutionary Algorithms, Online Evolution, Game AI, StarCraft

ACM Reference format:
Niels Justesen and Sebastian Risi. 2017. Continual Online Evolutionary Planning
for In-Game Build Order Adaptation in StarCraft. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071210

1 INTRODUCTION

This paper describes how an evolutionary-based approach, called *Continual Online Evolutionary Planning* (COEP), can control the macro-management tasks in StarCraft. Evolutionary algorithms have previously been applied to the problem of optimizing build orders [1, 15, 16], but only to the extent of optimizing fixed opening build orders, while COEP runs continually during the game (i.e. online) to adapt to the opponent. A StarCraft bot can be adaptive in two ways: It can be inter-game adaptive if it can change strategy between games to counter the playing style of the opponent and intra-game adaptive if it can adapt to the opponent's strategy within a game. Ontanón et al. conclude that "No bot is capable of observing the opponent and autonomously synthesize a good plan from scratch to counter the opponent strategy" [21] and as we see it both inter-game and intra-game adaptiveness have received little attention in the research community.

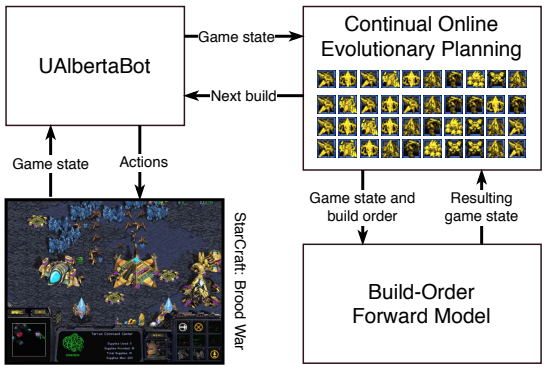

Figure 1: *Continual Online Evolutionary Planning* (COEP) continually evolves future build orders while a StarCraft bot (UAlbertaBot) executes the best one found so far.

This paper focuses on intra-game adaptiveness as, to our knowledge, no prior system exists that can perform in-game adaptive build-order planning for StarCraft. Our approach is unique as COEP runs continually to optimize the future build order while the game is being played, taking available information about the opponent's strategy into account. For the experiments in this paper we build on the modular UAlbertaBot, by replacing the module that is responsible for macro-management tasks (i.e. what builds to produce and in which order) with our evolutionary planner. Tasks such as controlling units in combat are performed by the UAlbertaBot itself and are in themselves an activate research area [8, 14, 26]. A series of experiments demonstrate that COEP can outperform the game's built-in bot as well as some scripted opening build-orders.

2 BACKGROUND

2.1 StarCraft

StarCraft is a real-time strategy (RTS) game released by Blizzard Entertainment in 1998. Its expansion set StarCraft: Brood War was released later the same year and became extremely popular as an e-sport. The sequel StarCraft II was released in 2010 with the same core gameplay but has a more modern interface as well as several new units and maps. This paper focuses on StarCraft: Brood War as it has gained the most popularity within the field of game AI, while the presented approach can be applied to all the games in the StarCraft series as well as similar RTS games.

GECCO '17, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3071178.3071210

Players control one of three races in StarCraft, Terran, Protoss and Zerg, each with their own strengths and weaknesses. Each player starts with four workers that can gather resources and construct buildings, as well as a base that can produce new workers. As the game progresses each player produces more advanced units, buildings, technologies, and upgrades (jointly referred to as *builds*) until one player is able to overrun the opponent's base. Advanced builds require that some basic builds are produced first and these requirements form a tree structure called a *tech tree*. A major part of a player's strategy is the order of builds produced, i.e. the *build order*, which determines the number and combination of units the player will have during the game.

StarCraft provides incomplete information about the game state, since the opponent's base is initially hidden and must be explored by scouting units. This, combined with the fact that multiple agents must be controlled in real-time, makes it a challenging environment for decision making agents. The decision-making process can be split into *micro-management* and *macro-management* tasks. We define **micro-management** as *the tactical control of individual units and buildings*, and **macro-management** as *the strategic planning of what builds to produce and in which order*. The approach introduced here only focuses on the macro-management tasks.

2.2 StarCraft Bots

The game has become an important benchmark in the field of game AI with several competitions such as the *AIIDE StarCraft AI Competition*[1], the *CIG StarCraft RTS AI Competition*[2] and the *Student StarCraft AI Competition*[3]. Many challenges must be overcome to succeed in these competitions, such as terrain analysis, pathfinding, and build order scheduling. However, as noted by Ontanón et al. in 2012, the most successful StarCraft bots rely mainly on hard-coded strategies [21], which is still the case today [9]. Many of these bots implement hard-coded build orders and are only able to adapt by following some predefined rules. The problem with hard-coded approaches is that the bot is limited to a fixed set of strategies, but more importantly the ability to adapt to what happens in the game is restricted as well. While a hard-coded approach can be successful against many other bots, it is easy for human players to counter these strategies.

Most StarCraft bots have a modular design, in which the tasks are divided into smaller sub-problems (i.e. a *divide and conquer* strategy). These modules often form hierarchy of abstraction that enables the top-level modules to perform macro-management tasks while lower level modules perform micro-management. The open source UAlbertaBot[4] by Churchill is an example of such an approach. The strategy manager maintains the strategy and communicates to the production manager what build order to follow. The production manager then takes care of assigning workers and buildings to produce the next builds in the queue, which happens simultaneously while the combat manager controls units in battle and the scout manager controls any scouting units. A strategy for UAlbertaBot can be described in a configuration file as a scripted build order; a hard-coded strategy followed by the bot. The modular design is described in more detail in Ontanón et al. [21].

StarCraft bots communicate with the game using the Brood War Application Programming Interface (BWAPI)[5]. BWAPI allows other C++ programs to access the game state in StarCraft: Brood War as well as giving commands to units, and is used by all the bots in the aforementioned competitions.

2.3 Build-Order Planning

There exist several approaches to build-order search and optimization for StarCraft. Churchill et al. implemented a depth-first branch & bound algorithm that finds the shortest possible time span to achieve a given goal (i.e. a list of units the build order should obtain [7]). The problem of optimizing opening build-orders has also been approached with multi-objective evolutionary algorithms [1, 15, 16] by encoding the genotype as a the list of builds. The strength of these methods is that they do not evolve build orders to reach one goal, but several. While these approaches to build-order optimization work well, even when compared to professional players, they are only designed to find an opening build-order and do not take the opponent's strategy into account as the game progresses. Synnaeve et al. show promising results for adaptive build-order planning; their approach can predict the opponent's strategy from the noisy observations in the game using a Bayesian model [25]. However their approach relies on hard-coded rules on top of the prediction model and it is unknown how well it will work when integrated into a bot. Another approach worth mentioning by Garćıa-Sánchez et al. [11] demonstrates how a complete strategy can be evolved, including both macro-management and micro-management behaviors. The evolved strategies are however static and do not change during a game. Some attempts have been made to predict the opponent's strategy from partial information [6, 25], but it has not been demonstrated how these approaches can be applied to build-order planning.

2.4 Online Evolutionary Planning

Evolutionary algorithms (EA) are a popular class of optimization techniques inspired by natural selection [12] that have been used for agents in various types of games. Online Evolutionary Planning (OEP) is a specific type of EA that can be applied to environments where an agent takes a number of actions sequentially. The algorithm's genotypes represent a sequence of actions and to evaluate the fitness of such a sequence (i.e. a plan), a forward model predicts the outcome of applying that plan in the current game state. A forward model can simulate the environment by implementing its rules or some abstraction of it. A heuristic then evaluates the resulting game state similarly to many tree search approaches.

Evolving behaviors for decision making agents using EAs has been a popular approach, for example for parameter-tuning of game-playing bots in FPS games [10]. Evolving neural controllers (i.e. neuroevolution) has also been applied in various contexts in games [23], including the racing competition TORCS [3, 4]. Usually EAs are applied in an *off-line* manner in which a behavior is evolved in a long series of training games. Recent work has however shown that EAs can also be successful for planning action sequences in an

[1] http://www.cs.mun.ca/~dchurchill/starcraftaicomp/
[2] http://cilab.sejong.ac.kr/sc_competition/
[3] http://sscaitournament.com/
[4] https://github.com/davechurchill/ualbertabot
[5] http://bwapi.github.io/

on-line manner. On-line evolutionary algorithms for decision making and planning, also called *Online Evolution*, *Online Evolutionary Planning* or *Rolling Horizon Evolution*, have been applied to the traveling salesman problem [22], two-player real-time games [17], the turn-based strategy game Hero Academy [13] and StarCraft combat situations [26]. The Online (1+1)-Evolutionary Algorithm has also been applied to evolve controller values online for a robot during the actual operation [2] and for a controller in a car racing game [20]. This algorithm differs from Online Evolutionary Planning as they evolve actuator values instead of action sequences. Tree search methods have limited success in such environments as the search tree needs to reach a certain depth to properly evaluate an action sequence (e.g. to properly evaluate a build order of 12 builds, the search tree must reach a depth of 12). One tree search method worth mentioning is Monte-Carlo Tree Search (MCTS) [5], which is based on random sampling and has been applied to games with large search spaces such as Go [24].

While OEP in itself is simple, depending on the environment it can be applied in various ways. Perez et al. applied the OEP variant *Rolling Horizon Evolution* to a real-time environment by evolving a series of short-term plans one by one [22] in a "rolling" fashion; while the current plan is being executed the proceeding plan is being evolved in parallel. This approach can react to changes within a real-time game, but only once the agent begins the next plan. The downside of this approach is that it must either evolve short plans or adapt slowly. The continual approach presented here always evolves plans from the current game state, which allows for faster adaptation, while still performing long-term planning.

3 APPROACH: CONTINUAL ONLINE EVOLUTIONARY PLANNING

Continual Online Evolutionary Planning (COEP) extends the original Online Evolution approach by Justesen et al. [13]. Each genome represents a candidate build order with a fixed length. To evaluate the fitness of genomes a *build order forward model* simulates the outcome of a build order (Section 3.1). The fitness function (Section 3.2) takes into account the resulting unit composition and available information about the opponent's units.

The most prominent difference to the original Online Evolution approach is that COEP runs *continually* in *parallel*. Additionally, when the bot requests a new build, it is taken from the build order of the currently most fit candidate (the champion) in the population. Simultaneously, the game state (*state* in Algorithm 1) is updated such that build orders are generated and evaluated based on a recent version of the game state. Furthermore, if builds have gone into production since the last update, genomes are updated such that the first instance of these builds are removed from their build order.

COEP runs a fixed number of generations after which it restarts using a new population. These restarts are intended to prevent the evolution getting stuck in local optima, which would prevent the system from adapting to the continuously changing game state. To avoid too much variance when new populations are created, the champion of the last population is transferred into the new population but excluded from the reproduction phase. In this way, COEP can attempt to evolve new build orders while keeping the best from the last population until a superior solution is found.

Algorithm 1 Continual Online Evolutionary Planning (COEP)

COEP continually creates a new population and runs evolution for number of generations. *State* is updated by the bot as soon as new information is obtained and the best found build order can be retrieved from the *champion* genome.
procedure COEP(GameState *s*) ▷ *s* is the initial game state.
  champion = NULL ▷ Accessible by bot
  state = *s* ▷ Accessible by bot
  while game is not over **do**
    pop = ∅ ▷ Create new population
    if *champion* is not NULL **then** *pop*.Push(*champion*)
    for *i* = Size(*pop*) **to** POP_SIZE **do**
      g = Genome(*s*)
      g.buildOrder = legal build order from *s*
      g.fitness = FITNESS(*s*, *genome*.buildOrder)
      pop.Push(*genome*)
    for *i* = 1 **to** GENERATIONS **do**
      Reduce *pop* based on elitism rules
      Reproduce offspring using crossover
      Mutate some offspring
      Evaluate fitness of offspring
      Add offspring to *pop*
      champion = most fit genome

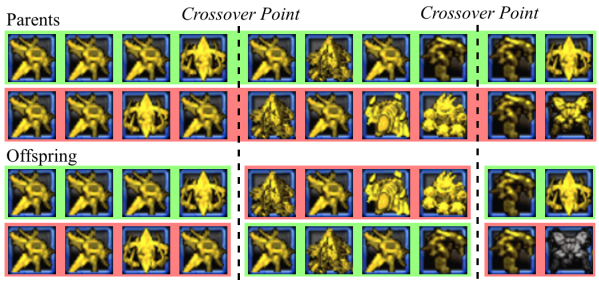

Figure 2: Two-point crossover for two parent build orders and the resulting offspring. Notice that the build in the bottom right corner remains in the genotype but becomes inactive because its requirements are no longer met.

This idea is similar to case-injected genetic algorithms, in which solutions to previously solved problems are periodically injected into the population [18, 19].

Crossover is applied directly to build orders from two randomly sampled parents (Figure 2). Computational resources on the forward model are limited by checking legality only when genomes are generated for a new population, and not after crossover and mutation. Thus, it is important to trim illegal builds when the bot request a build order. If some builds within a build order of an offspring are illegal, the forward model simply ignores them.

Four different mutation operators make sure that existing build orders can be reorganized effectively and new genes with several requirements can be introduced (Figure 3): **Clone:** Two indices *a* and *b* are randomly selected. Build at position *a* becomes the same as the build at position *b*. **Swap:** Two random builds swap position. **Add:** One random build is randomly inserted. For each

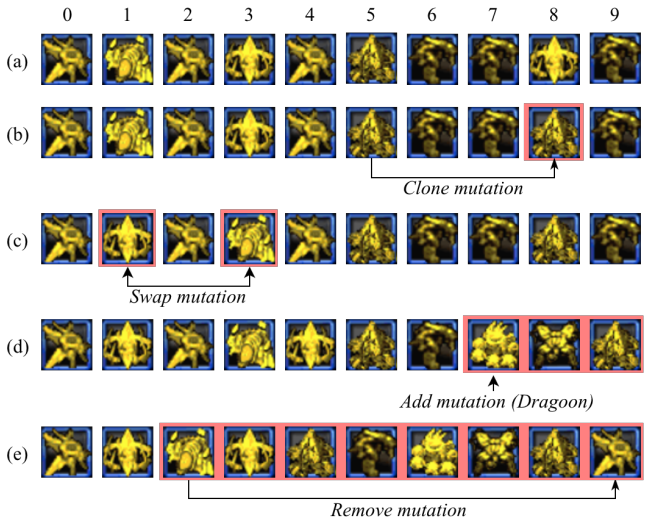

Figure 3: A build order with ten builds, which is manipulated by the four mutation operators. Builds are highlighted (red) if they are changed during an operation. (a) Shows the initial build order, (b) the result of a clone mutation from index 5 to 8, (c) a swap mutation on index 1 and 3, which swaps the two builds, (d) an add mutation on index 7, which adds a dragoon and recursively adds its requirements first and (e) a remove mutation at index 2 that moves the build to the end of the build order.

unmet requirement, each required build is recursively added in front, such that the first build is moved backwards and eventually out of the build order. **Remove:** One random build is moved to the end of the build order and all builds after the moved build's initial position slide one step forward.

3.1 Forward Model

A forward model can predict the outcome of taking some actions in a given game state, which is necessary to evaluate the fitness of build orders. In this paper, the model does not need to implement all the game rules since we are only concerned with macro-management and not how units move around the map. Such a *build order forward model* was implemented for the Protoss and Terran race and the source code is available[6]. The forward model (Algorithm 2) iterates the given build order and tries to add each build to the given game state in order; if the requirements of a build are not satisfied it is simply ignored.

A few constants are used by the forward model: MINE_SPEED and GAS_SPEED refers to the amount of minerals/gas one worker gathers on average in one frame and has been estimated to 0.05 and 0.07 respectively (similar values of 0.045 and 0.07 were used by Churchill et al. [7]). The amount of minerals gathered decreases if more than ten workers mine at each base, such that workers 11–20 only gather half as many minerals, 21–30 a third, etc.

COEP receives information available about the current game state as input, which includes the number of all known friendly and enemy units, buildings, technologies and upgrades as well as the current frame number. The technologies and upgrades the opponent has researched are not known so they are excluded. Also note that the game state only includes the partial knowledge about the enemy units that the player has obtained. Additionally, the game state includes a list of friendly builds that are in production as well as the frame number in which they are completed.

[6]https://github.com/njustesen/coep-starcraft/

Algorithm 2 StarCraft Build-Order Forward Model
MineralWorkers(s) and GasWorkers(s) returns the number of workers gathering minerals and gas, respectively, in s.

```
1:  Predict(s, buildOrder, endFrame) returns the resulting game
    state of producing the builds in buildOrder from state s until
    frame endFrame is reached.
2:  procedure Predict(GameState s, BuildType[] buildOrder, int
    endFrame)
3:      for each BuildType type in buildOrder do
4:          nextFrame = ProduceFrame(s, type)
5:          if nextFrame ≤ endFrame then
6:              Progress(s, nextFrame)
7:              Build(s, type)
8:          else
9:              Progress(s, endFrame)
10:     return s                          ▷ The altered game state.
11: procedure ProduceFrame(GameState s, BuildType type)
12:     Returns the latest frame in which all requirements, re-
        sources, and production buildings/units are available in s in
        order to produce type.
13: procedure Progress(GameState s, int toFrame)
14:     t = toFrame - s.frame
15:     s.minerals += t × MINE_SPEED × MineralWorkers(s)
16:     s.gas += t × GAS_SPEED × GasWorkers(s)
17:     for each Build b in s.underProduction do
18:         if not b.done and toFrame ≥ b.doneAt then
19:             Add one build of type b.type to s
20:             b.done = true
21: procedure Build(GameState s, BuildType type)
22:     b = Build()
23:     b.type = type
24:     b.doneAt = s.frame + type.buildTime
25:     s.underProduction.Push(b)
26:     s.minerals -= type.mineralCost
27:     s.gas -= type.gasCost
28:     s.supply += type.supplyCost
```

3.2 Fitness

Building on the forward model, which predicts the resulting game state after applying a build order, the fitness of a build order is determined by how desirable this future game state is for the player. A challenge with this naive approach is that, at least in real-time games, the longer one tries to predict into the future the more uncertain the outcome becomes. For example, a build order with a very strong economy in the beginning and a large unit production in the end would give a high fitness, even though the player has no army and is defenseless during most of the evaluated period.

Algorithm 3 Discounted Accumulated Fitness

```
1: Sets the fitness of a genome by calculating the discounted
   accumulated fitness of several steps of stepSize frames for a
   total of horizon frames.
2: procedure Fitness(Genome g, GameState s, int horizon, int
   stepSize)
3:     state = Clone(s)
4:     step = 0
5:     while state.frame < s.frame + horizon do
6:         next = Min(s.frame + horizon, state.frame + stepSize)
7:         build = next unbuilt build in g.buildOrder
8:         state = Predict(state, [build], next)
9:         g.fitness += Heuristic(state) ×DISCOUNT^step
10:        step += 1
```

	Zealot	Dragoon	Dark Templar	Scout
Marine	0.7	0.5	0.6	1.6
Firebat	1.3	0.1	0.8	0.7
Vulture	1.6	0.7	1.6	0.7
Goliath	0.9	0.7	0.9	1.5
Siege Tank	0.8	1.4	0.5	0.9

Table 1: Unit matchup table that values how strong units are against each other which is a critical part of the heuristic applied. Values are in the range [0, 2].

Therefore, the fitness function introduced in this paper performs an evaluation several times during the time span of the build order in addition to incorporating a discount factor (inspired by the Bellman equation) that values short-term rewards higher than long-term rewards. In other words, instead of applying the forward model one time on the entire build order, it is applied in several steps on subsets: within each step the heuristic of the intermediate game state is accumulated into the final fitness of the build order (Algorithm 3).

The fitness function is based on a heuristic that can evaluate how desirable game states are. The game state in StarCraft is seen from a player perspective and is thus only partial visible (see Section 3.1). Designing an optimal heuristic for StarCraft is extremely challenging and highly dependent on the applied micro-management policy. The simple heuristic in this paper evaluates players' unit composition based on specific StarCraft domain knowledge. Some units are superior against particular units while inferior against others (e.g. the powerful Zerg ultralisk, a ground melee unit, is defenseless against a Protoss scout, a flying ranged unit). To express how strong each unit type is against any other unit type a *unit matchup table* is introduced (Table 1). For example, the Terran firebat (short-ranged unit) is valued 0.4 against a Protoss dragoon (long-ranged unit) to express its weakness in this matchup. The value of a dragoon against a firebat is the same, but inverted: $2 - 0.4 = 1.6$. Attributes such as damage types, unit size and whether they are invisible or can detect invisible units are also considered.

Upgrades and technologies can improve the strength of some units, which is reflected in Table 2. The values in this table are multiplied with the matchup value from Table 1 to determine the final values. For example, a Protoss dragoon has a final value against a Terran firebat of $1.6 \times 1.25 = 2$, if the *Singularity Charge* upgrade has been researched. Note that upgrade bonuses are not added to enemy units. We define a function *matchup* that performs these calculations given a friendly unit type x and enemy unit type y (e.g. *matchup*(dragoon, firebat) = 1.6; *matchup*(firebat, dragoon) = 0.4). The value for player p of a unit matchup of friendly units of type x and enemy units of type y is:

	Zealot	Dragoon	Scout
Ground Armor	1.02	1.02	-
Plasma Shields	1.02	1.02	1.02
Air Armor	-	-	1.02
Singularity Charge	-	1.25	-

Table 2: Upgrade and tech multipliers, which give units additional value in the heuristic.

$$value(p, x, y) = matchup(x, y) \times n(x) \times n(y) \times \left(1 - \frac{n(x)}{N(p)}\right),$$

where $matchup(p, x, y)$ refers to the unit matchup table, $n(y)$ and $n(x)$ is the number of units of type y and x, and $N(p)$ is the number of all units controlled by player p. The idea is that a player should strive to optimize all four components of this function to achieve a good unit combination. This heuristic prefers a balanced unit composition, in which units individually have high unit matchup values against the enemy units. The first three components increase if the player has many units that counter the enemy units while the last component $(1 - \frac{n(x)}{N(p)})$ increases if the player has a balanced mix of unit types. It should be noted that $n(x)$ in the last component is further divided by 2 if x is a worker. The final heuristic for calculating the discounted accumulated fitness of a given state S (Algorithm 3) with players $p1$ and $p2$ is the sum of all values for all permutations of both players' units types u_{p1} and u_{p2}:

$$heuristic(S) = \sum_{x}^{u_{p1}} \sum_{y}^{u_{p2}} value(p1, x, y) - value(p2, y, x).$$

After prior experimentation with this heuristic we found it necessary to penalize expansions while having few workers as well as not expanding while having many workers. The expansion penalty in state s is equal to $numOfBases \times 14 - \text{MineralWorkers}(s)$. Likewise, a penalty for having too many supply buildings was found necessary. A complete implementation of the heuristic can be found in the source code[7].

3.3 Integration with UAlbertaBot

Because of UAlbertaBot's modular design it is simple to replace the existing production manager module with the presented COEP approach; all other modules in the bot are kept unchanged. The new production manager now requests the COEP for a build whenever a new build is being produced or if 600 frames have passed. Since our implementation of COEP is in Python, the production manager implements an HTTP client and communications with COEP through an HTTP server using the Django framework. This setup works well for experimentation but with the downside that it cannot run in BWAPI's release mode and thus not as part of the current AI competitions. The entire setup is composed of UAlbertaBot running in one process which communicates with another process that is divided into two threads, one for Django and one for COEP.

[7] https://github.com/njustesen/coep-starcraft

4 EXPERIMENTS

This section is split into two parts, where the first part consists of experiments that tests the ability of Online Evolutionary Planning (OEP), without the continual extension, to evolve strong build orders for static game states in StarCraft. Without the continual extension, the algorithm runs normally for a fixed number of generations using the same game state and then terminates. In the second part, COEP is applied to UAlbertaBot and is then tested in a total of 900 StarCraft games against the game's built-in bot as well as UAlbertaBot with four scripted opening build orders. The games were played on the two-player map Astral Balance and all the game replays are made available[8].

The population size of the algorithm is set to 64 with a survival rate of 25%. A two-point crossover operator is employed, and mutation operators (clone, swap, add and remove) are each activated individually with a 50% probability. This combination of mutation operators was found to perform best among the tested configurations (see Figure 4). For the fitness function a step size of 2 minutes (9,810 frames) is used, and a horizon of 8 minutes (11,429 frames), resulting in genomes with a build order length of 57 builds (200 frames per build), and a discount factor of 0.9. It takes on average 156±18 ms. for the algorithm to run one generation on a regular laptop (2,6 GHz Intel Core i5).

4.1 Online Evolutionary Planning Results

Online Evolutionary Planning (OEP) was tested on its ability to evolve build orders to counter different enemy unit combinations. More specifically, OEP had to find effective build orders for a Protoss player against a Terran player. Six different scenarios were created (Table 3) all with one nexus (main base), four probes (workers) and one pylon (supply building) for the Protoss player, each with a different set of units for the Terran player. For each of the six scenarios, OEP ran for 100 generations with a horizon of 12 minutes. Table 3 shows the unit combination of the best evolved build orders averaged over 50 independent evolutionary runs. The results demonstrate that OEP is capable of evolving diverse unit combinations that clearly depend on the combination of enemy units. For example, in the scenario shown in row six, the algorithm avoids zealots and dark templars (both ground melee units) against wraiths and battlecruisers (both flying units). In scenario 2 the algorithm prefers dragoons (long-ranged units) against firebats (short-ranged units). The reason why zealots and dragoons are so dominant in the evolved build orders is that they are cheaper units that can be produced early in the game. Referencing the values in the unit matchup table (Table 1) shows that the evolved build orders produce matching unit combinations.

To determine the importance of the introduced mutation operators, we ran 50 independent evolutionary runs for 100 generations with only one of the four mutation operators enabled, compared to all of them enabled (Figure 4). Interestingly, the clone and swap operators where the most efficient, but significantly less effective than as all four operators together ($p < .01$; two-tailed Mann-Whitney U Test). The algorithm was also tested with uniform crossover, single-point crossover and two-point crossover, but no significant change in performance was detected.

[8] http://bit.ly/2omfT5G

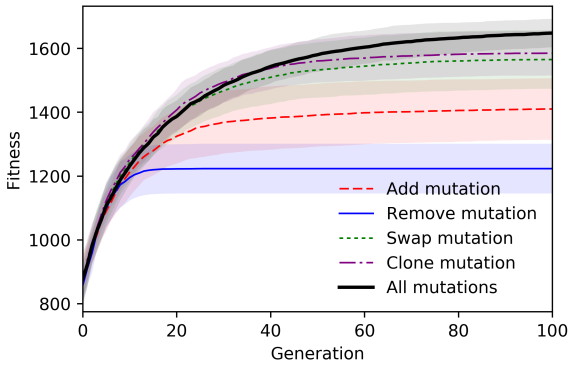

Figure 4: The average fitness over generations for Online Evolutionary Planning using a different mutation operator. Opaque coloring shows standard deviations.

4.2 Continual Online Evolutionary Planning Results

In the previous experiments, it was shown that OEP is capable of evolving build orders to counter the opponent's strategy. In the following experiments COEP is applied to UAlbertaBot to perform in-game build-order planning, playing as the Protoss race. COEP uses the same configuration as in the previous experiment as well as 100 generations in each loop. The bot played a total of 300 games against the built-in bots in StarCraft, 100 against each of the three races. Our bot won 275 games (91.7%) with 5 games (1.7%) ending in a draw. A summary of the results can be seen in Table 4. Each iteration of COEP, which consists of initializing a new population and running 100 generations, takes on average 9.96±0.8 seconds. COEP was also tested with a random fitness function, which performs significantly worse, corroborating the heuristic chosen in this paper. In most games the bot demonstrated the ability to adapt to the opponent's strategy efficiently enough to win. An example of such adaption is shown in Figure 5. The upper plot displays the number of zealots, dragoons, marines and firebats in the game. It is clear that our system (controlling the Protoss units) prefers zealots against the enemy marines but switches to a unit combination dominated by dragoons when firebats are spotted. This adaption rule can be seen in Tables 3 and 1. The bottom plot shows the highest fitness in the population over time as well as the times the COEP's game state was updated.

The final experiment compared our adaptive approach with four scripted protoss strategies played by UAlbertaBot. This test is more challenging as these scripts employ established opening strategies, optimized to destroy the enemy early in the game. Zealot / dragoon / DT rushes are aggressive strategies, in which the player tries to obtain an army of only one type of unit (zealots, dragoons or dark templars) as fast as possible to surprise the opponent. Still COEP was competitive against these challenging openings, winning 52% of all games (draws counted as half a win). The most challenging for COEP were the very fast zealot rush, which did not give much time to adapt. These results are summarized in Table 5.

Terran units								Average unit combinations of evolved build orders.					
SCV	Marine	Firebat	Vulture	Goliath	Siege Tank	Wraith	Battle-cruiser	Zealot	Dragoon	Dark Templar	Reaver	Scout	Carrier
10	**10**	0	0	0	0	0	0	**7.5** ± 2.8	**8.2** ± 4.2	**1.2** ± 2.0	**2.3** ± 2.3	**0.0** ± 0.0	**0.1** ± 0.4
10	0	**10**	0	0	0	0	0	**2.2** ± 2.0	**11.8** ± 5.0	**0.5** ± 0.9	**2.3** ± 2.6	**0.4** ± 0.7	**0.3** ± 0.9
10	0	0	**8**	0	0	0	0	**1.0** ± 1.0	**7.5** ± 3.5	**0.0** ± 0.1	**1.6** ± 2.1	**1.2** ± 1.5	**1.4** ± 1.6
10	0	0	**4**	0	**4**	0	0	**2.9** ± 1.9	**3.5** ± 2.7	**0.3** ± 0.1	**2.6** ± 2.9	**0.6** ± 1.1	**1.4** ± 1.6
10	0	0	0	**4**	**4**	0	0	**6.1** ± 3.0	**2.8** ± 2.8	**0.6** ± 1.8	**3.2** ± 2.7	**0.1** ± 0.4	**0.6** ± 1.0
10	0	0	0	0	0	**4**	**2**	**0.8** ± 1.0	**9.7** ± 3.7	**0.0** ± 0.3	**1.6** ± 1.6	**0.4** ± 0.7	**0.6** ± 1.0

Table 3: Unit combinations of evolved build orders found by Online Evolutionary Planning after 100 generations. Results are averaged over 50 evolutionary runs. Some units are excluded from the results for brevity. Each row represents one scenario containing the Terran units on the left as well as a Protoss nexus, pylon and four probes. The Protoss units on the right are the average unit combination of the evolved build orders. For each unit type, the average count as well as the standard deviation is shown. The main result is that by following the implemented heuristics, Online Evolutionary Planning is able to evolve build orders that can effectively counter the opponent's strategy.

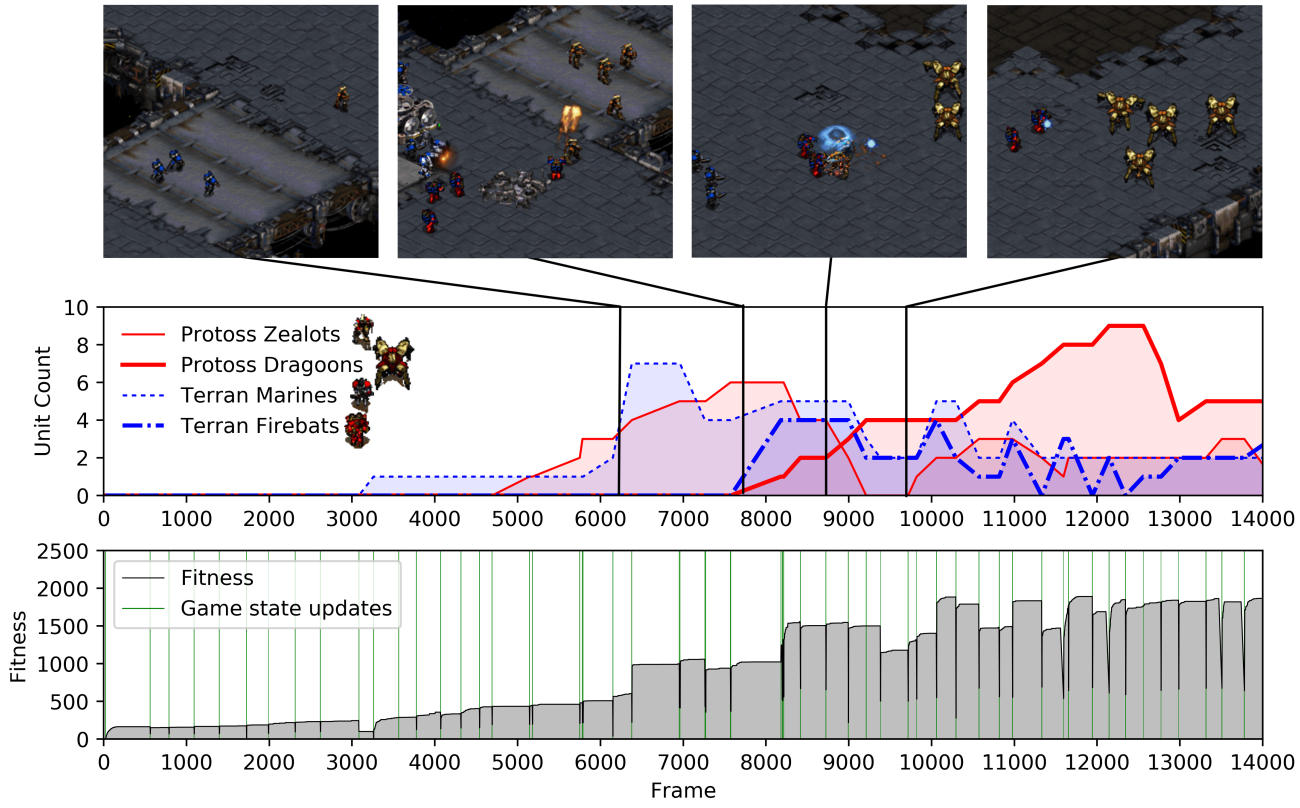

Figure 5: A visualization of Continual Online Evolutionary Planning's (COEP) ability to adapt a Protoss build order in-game against the built-in Terran bot. The upper plot shows the number of zealots, dragoons, marines and firebats over time and the lower plot shows the highest fitness in the population. Green vertical lines indicate when the game state was updated. The four screenshots in the top show critical situations in the game. Early in the game the bot observes a group of Terran marines and continues to produce zealots to counter them. Shortly after, these zealots fight against a large group of Terran firebats and many zealots die. COEP quickly adapts its strategy to switch production to dragoons as they are superior to firebats. A video of this game can be found here: https://youtu.be/SCZbDpIaqmI.

	Protoss	Terran	Zerg
COEP	83/4/13	96/0/4	96/1/3
COEP Random Fitness	1/0/99	0/0/100	4/0/96

Table 4: Number of wins, draws and losses by Continual Online Evolutionary Planning (COEP) against each of the three races controlled by the built-in bot in StarCraft. The bottom row shows results of COEP with a random fitness function.

	Zealot Rush	Dragoon Rush	DT Rush
COEP	19/0/81	60/0/40	73/7/20

Table 5: Number of wins, draws and losses by Continual Online Evolutionary Planning (COEP) against three scripted Protoss opening strategies performed by UAlbertaBot.

5 DISCUSSION

In some cases COEP can struggle, such as when it has to adapt to the very aggressive zealot rush. Since our heuristic only takes the enemy units, and not production buildings into account, COEP's ability to adapt is delayed, which is devastating during rushes. In the future we plan to extend COEP to also take buildings into account.

Designing the heuristic has been challenging as it needs to correlate with the playing style of the underlying bot. UAlbertaBot implements a specific behavior, which has its own quirks when it comes to controlling larger groups of units or when it expands to new bases. The strategies preferred by our implementation involve large armies with various unit types which require more advanced micro-management compared to the simpler rush strategies. UAlbertaBot also displayed difficulties using more advanced units such as reavers, high templars and shuttles, which limits the range of possible unit combinations for our approach. UAlbertaBot was probably not designed to have an adaptive build order module which requires a great deal of generality in its implementation. Developing a more advanced and general StarCraft bot, or improving upon an existing bot, as well as fully incorporating COEP are important next steps.

Instead of having a complete reactive approach it might be fruitful to imagine what the opponent is doing along with our own planning. Introducing co-evolution by also evolving build-orders for the opponent player, could perhaps provide a more preventive behavior.

6 CONCLUSIONS

This paper presented a variation of Online Evolutionary Planning called *Continual Online Evolutionary Planning* (COEP) that can perform adaptive build order planning in StarCraft. COEP implements a discounted accumulated fitness function that favors short-term rewards over long-term rewards. COEP was applied to an existing StarCraft bot called UAlbertaBot, where it replaced the existing macro-management module. The results demonstrate that COEP is capable of in-game build-order planning, continually adapting to the changes in the game. While COEP still struggles against some very aggressive rushes, it outperforms the built-in bot in StarCraft: Brood War with a 91.7% win rate and can compete with a number of scripted opening build orders performed by UAlbertBot.

REFERENCES

[1] Jason Blackford and Gary B Lamont. 2014. The Real-Time Strategy Game Multi-Objective Build Order Problem.. In *AIIDE*.

[2] Nicolas Bredeche, Evert Haasdijk, and AE Eiben. 2009. On-line, on-board evolution of robot controllers. In *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 110–121.

[3] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. 2009. On-line neuroevolution applied to the open racing car simulator. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE, 2622–2629.

[4] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. 2010. Applying cooperative coevolution to compete in the 2009 torcs endurance world championship. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE, 1–8.

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-Carlo Tree Search: A New Framework for Game AI.. In *AIIDE*.

[6] Ho-Chul Cho, Kyung-Joong Kim, and Sung-Bae Cho. 2013. Replay-based strategy prediction and build order adaptation for StarCraft AI bots. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 1–7.

[7] David Churchill and Michael Buro. 2011. Build Order Optimization in StarCraft.. In *AIIDE*. 14–19.

[8] David Churchill and Michael Buro. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 1–8.

[9] David Churchill, Mike Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontanón, and Michal Certickỳ. 2016. Starcraft bots and competitions. (2016).

[10] Nicholas Cole, Sushil J Louis, and Chris Miles. 2004. Using a genetic algorithm to tune first-person shooter bots. In *Evolutionary Computation, 2004. CEC2004. Congress on*, Vol. 1. IEEE, 139–145.

[11] Pablo Garćıa-Sánchez, Alberto Tonda, Antonio M Mora, Giovanni Squillero, and JJ Merelo. 2015. Towards automatic StarCraft strategy generation using genetic programming. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*. IEEE, 284–291.

[12] H Jo. 1975. Holland:" Adaption in Natural and Artificial Systems. (1975).

[13] Niels Justesen, Tobias Mahlmann, and Julian Togelius. 2016. Online evolution for multi-action adversarial games. In *European Conference on the Applications of Evolutionary Computation*. Springer, 590–603.

[14] Niels Justesen, Bálint Tillman, Julian Togelius, and Sebastian Risi. 2014. Script- and cluster-based UCT for StarCraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 1–8.

[15] Harald Köstler and Björn Gmeiner. 2013. A multi-objective genetic algorithm for build order optimization in StarCraft II. *KI-Künstliche Intelligenz* 27, 3 (2013), 221–233.

[16] Matthias Kuchem, Mike Preuss, and Günter Rudolph. 2013. Multi-objective assessment of pre-optimized build orders exemplified for starcraft 2. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 1–8.

[17] Jialin Liu, Diego Pérez-Liébana, and Simon M Lucas. 2016. Rolling Horizon Coevolutionary Planning for Two-Player Video Games. *arXiv preprint arXiv:1607.01730* (2016).

[18] Sushil J Louis and John McDonnell. 2004. Learning with case-injected genetic algorithms. *IEEE Transactions on Evolutionary Computation* 8, 4 (2004), 316–328.

[19] Sushil J Louis and Chris Miles. 2005. Playing to learn: Case-injected genetic algorithms for learning to play computer games. *IEEE Transactions on Evolutionary Computation* 9, 6 (2005), 669–681.

[20] Samadhi Nallaperuma, Frank Neumann, Mohammad Reza Bonyadi, and Zbigniew Michalewicz. 2014. EVOR: an online evolutionary algorithm for car racing games. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 317–324.

[21] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. 2013. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games* 5, 4 (2013), 293–311.

[22] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. 2013. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 351–358.

[23] Sebastian Risi and Julian Togelius. 2015. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games* (2015).

[24] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and others. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[25] Gabriel Synnaeve and Pierre Bessiere. 2011. A Bayesian model for plan recognition in RTS games applied to StarCraft. *arXiv preprint arXiv:1111.3735* (2011).

[26] Che Wang, Pan Chen, Yuanda Li, Christoffer Holmgård, and Julian Togelius. 2016. Portfolio Online Evolution in StarCraft. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*.