Search Ordering for StarCraft Build Order Optimization

Benjamin Stanley
Department of Computer Science
Memorial University
St. John's, Canada
Email: bstanley@mun.ca

David Churchill
Department of Computer Science
Memorial University
St. John's, Canada
Email: dave.churchill@gmail.com

***Abstract*—In real time strategy games, optimizing the order in which units and other items are built is essential to high level play. A sequence of such build actions is referred to as a build order. In StarCraft, pro players have developed many build orders for specific situations, but the task of creating an optimized build order given an arbitrary goal remains open. We approach this problem using the depth first search branch and bound algorithm, which uses an upper bound to decrease search time. Ordering the search in different ways has the potential to lower this bound earlier, speeding up the algorithm. We devised 17 such ordering methods and compared their performance across 10 build order goals. We show that this technique, depending on the chosen ordering, significantly reduces the search time. This provides an advantage in a real time game setting.**

***Index Terms*—Video Games, Heuristic Search, Artificial Intelligence, Planning.**

I. Introduction

A. StarCraft

StarCraft is a real time strategy game developed by Blizzard Entertainment. Players choose from three factions: Zerg, Terrans, and Protoss, each with their own distinct units and mechanics. Each player starts with a handful of units with which they must collect resources and build up an army to crush their opponents. When devising a strategy, a player must consider the strengths and weaknesses of their faction, the factions chosen by their opponents, and the layout of the map, among other factors. As a result, developing an effective strategy to win a game of StarCraft is highly challenging.

StarCraft's strategic depth attracted the attention of artificial intelligence researchers, who identified the game as a suitably complex environment to develop and test new models. Research into StarCraft revolves around designing and improving upon bots that can play the game against each other or humans. Multiple tournaments have been organized for such bots, such as the Artificial Intelligence and Interactive Digital Entertainment (AIIDE) StarCraft AI Competition or IEEE's Computational Intelligence and Games (CIG) StarCraft RTS AI Competition [1]. For a detailed review of these competitions and the history of early research in this field we refer the reader to [1].

B. Build Order Optimization

Artificial intelligence systems made for StarCraft must account for numerous aspects of the game, from unit micromanagement to army composition. In this research we focus on one such aspect: build order optimization.

In StarCraft, each unit and building has requirements that must be met before they can be made, usually the construction of a certain building. These requirements form an implicit tech tree that the player must progress through to unlock high level units. The player must also manage the number of worker units necessary to build buildings and collect the game's two resources: gas and minerals. Each race also has unique mechanics for construction, such as the Zerg needing to consume a worker unit for each building they make. Furthermore, StarCraft being a real time game, multiple actions can be taken at once and each takes time to complete. When combined, these factors make the task of planning construction quite difficult. That is where build order optimization takes the stage.

A build order is a series of actions related to construction that a player takes in sequence. The types of actions contained within a build order (build actions) are those that cost resources and time to produce: either a unit, building, technology, or upgrade. Upon completion of a build order, the player will have achieved some goal in terms of army composition or base construction.

Our research focuses on a specific form of build order: a series of actions that result in the completion of a given goal. The goal is a set of build actions that must be completed in a given state. For example, a goal could be "Five Hydralisks", which would be fulfilled by a state where the player has built five Hydralisk units.

In this context an optimal build order is one that produces the goal in the shortest makespan, or total time. Quickly finding optimal build orders gives a strategic edge in real time strategy games like StarCraft. Thus, it is imperative to minimize the computational time required to find an optimal build order. The build order optimization we present here is the process of doing just that: devising efficient methods to find optimal build orders given a user-defined goal.

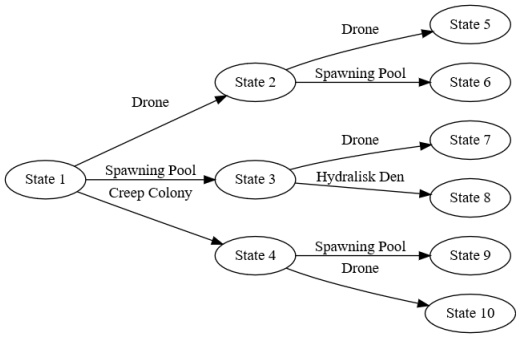

Fig. 1: A tree of game states that represents some initial Zerg build orders.

C. Heuristic Search

The primary method we use for build order optimization is heuristic search. Heuristic search is the process of navigating the possible states of a system, called the search space, through use of heuristics. The heuristics can guide the search and limit the size of the search space that must be considered. Search can be represented as graph traversal, and uses algorithms such as depth-first search, best-first search, and A* search.

Applying search to video games implies taking each possible state of the game as a vertex, and the actions that transition the game between these states as edges. In the context of StarCraft build orders, the edges between states are build actions and the states record what the player has built, their resources, and the current time. It is thus possible to construct a tree where the path from one state to another is the build order required to transition between them. These infinite trees are the space in which the search for the optimal build order is conducted.

Making this infinite space efficiently searchable is the role of the heuristics. The goal of our research is to develop a heuristic method to cut down the size of this search space as much as possible.

D. Build Order Search System

This paper follows research by Churchill and Buro [2] that deployed the depth first branch and bound (DFBB) algorithm to search for optimal build orders. To run this search they developed the Build Order Search System (BOSS), a simulator of StarCraft's build mechanics. StarCraft is a video game that runs in real time, with many mechanics besides building. It is not possible or at all desirable to perform a search using the game itself, so a simulator such as BOSS is necessary.

BOSS neglects all aspects of the game not strictly necessary for build order simulation. It accounts for only a single player and ignores irrelevant mechanics like combat. It abstracts away decisions such as worker management or building placement such that the only actions the system accepts are build actions. BOSS also accounts for resource collection and undertaking multiple actions simultaneously. This makes BOSS easy to use and much faster than the real game. The computational efficiency this affords makes search possible.

E. StarData

The other project we use in our research is StarData, a database of StarCraft replays compiled by researchers at Facebook [3] (now Meta). StarCraft replays are recordings of all actions taken within a match. They are produced and replayed through StarCraft itself. Thousands of these replays have been uploaded online, forming a database of knowledge on how StarCraft is played. The problem is that replays record low level actions such as clicks, not high level actions like building. This necessitates running the replays in the game itself to analyse them, and also sometimes results in replays recorded on one version of the game being unreadable on others. Processing this data in an efficient manner is nearly impossible.

To remedy this, Facebook's researchers collected a set of 65646 replays of real human games and went through the trouble of running them all. As each one ran, they converted it into a format more conducive to data analysis. They gave the resulting dataset the name StarData [3].

StarData is a part of a larger StarCraft project called TorchCraft, which acts as an interface between the machine learning library Torch (also PyTorch) and StarCraft [4]. StarData was created using TorchCraft and must be read through TorchCraft. We had no intention of using TorchCraft in our research, so we developed a standalone program to extract information from StarData containing only those parts of TorchCraft that were strictly necessary to read replays. We use that extractor in this research to get data on which to base new heuristics for search.

II. Methods

We base our algorithm on Churchill and Buro's depth first branch and bound approach. DFBB is identical to normal depth first search, except it applies an upper bound. In this case the upper bound is the makespan of the best solution encountered so far in the search. Any potential build orders whose makespans exceed the upper bound are discarded. This bounds the otherwise infinite search space, making depth first search possible.

In depth first search, the order in which adjacent vertices are visited is usually arbitrary. For depth first branch and bound however, the order becomes significant. If a DFBB search encounters a better solution earlier, the upper bound will be lower, culling more search nodes, and saving more time. As a result, changing the order in which the algorithm considers actions has the potential to speed it up by finding better solutions [5]. For example, a perfect ordering that always chooses optimally would find the best build order first and render the search linear time rather than exponential. We devised 17 ordering methods and compared their performance in various scenarios to demonstrate the effectiveness of this technique.

A. Naive Build Order

Before listing the search orderings it is necessary to explain another method we use: naive build orders. StarCraft's build

mechanics are completely known, such that it is a trivial task to find a build order from any state to any valid goal. These naive build orders are rarely optimal, but they are very useful in finding the optimal solution. Before starting the search, determining a naive build order can provide an initial lower bound that limits the search space. Naive build orders can also be used for search ordering, as we will show.

Deriving naive build orders for StarCraft involves identifying all the goal's prerequisites on the tech tree, how many workers are needed to build everything, and accounting for other mechanics such as supply. This process runs in linear time relative to the prerequisites, providing sufficient build orders in a short time frame.

B. Search Orderings

Search orderings sort the actions that are legal at each step of the search. The orderings we devised are: ID-Increasing and Decreasing; F1-4; Naive Preferred ID-Increasing and Naive Preferred F2; Most amd Least Prerequisites; Greatest and Least Mineral + Gas Cost; Longest and Shortest Build Time; Latest and Earliest Completion Time; and Random. These orderings are all listed in Table I.

ID-Increasing and ID-Decreasing are based off the order in which the actions are stored in the system. ID-Increasing is effectively the default ordering, as a depth first search iterating over actions arbitrarily in the order they are stored will follow it. ID-Decreasing is the reverse.

The F1, F2, F3, and F4 orderings are based on frequency data extracted from StarData. Our intention with these orderings was to guide the search based off real players' build orders. StarData contains full StarCraft replays from which we were able to extract a wide variety of build orders. We sorted the actions from these build orders into bins based on their type and the time at which they took place, in 500 frame intervals. This created a two-dimensional matrix of each action's frequency data. A cell in this matrix is represented as M_{ij} where i indicates the type of the action and j indicates the time interval.

From this dataset we derive four metrics, labeled F1, F2, F3, and F4. Each of these metrics serves as the basis for its own ordering scheme.

$$\text{F1} = \frac{M_{ij}}{\sum_n M_{nj}}$$

F1 is the ratio of a build action's appearances in a given interval to the total number of build actions, of all types, taken during that interval.

$$\text{F2} = \frac{M_{ij}}{\sum_n M_{in}}$$

F2 is the ratio of a build action's appearances in a given interval to the total number of times that action was taken across all intervals.

$$\text{F3} = \text{F1} \cdot \text{F2}$$

$$\text{F4} = \text{F1} \cdot \log \frac{1}{\text{F2}}$$

Both F3 and F4 are attempts to combine the effects of F1 and F2.

These orderings compute the metric values for each action based on its type and the game state's current time, then ranks them from greatest to least.

The Naive Preferred methods are alterations on ID-Increasing and F2 that run a naive build order from the current state to the goal, then move the first action in that build order to the front of the list. The other actions are then sorted according to the base ordering. The idea is to first check the most obvious option, as that is often correct. ID-Increasing was chosen as a baseline, and F2 was chosen due to it performing well in early tests.

Besides random, the rest of the orderings are based off the inherent properties of the actions, each with greatest to least and least to greatest versions.

The Most Prerequisites and Least Prerequisites orderings were devised as a way of considering how "High Tech" an action is. These orderings sort actions based off the sum of the recursive prerequisites of an action. For example the Zerg Lair building has as its prerequisite the Spawning Pool, which has as its prerequisite the Hatchery, which has no prerequisite. The Lair thus has two total prerequisites and is sorted based on that.

Greatest Mineral + Gas Cost and Least Mineral + Gas Cost sort based on the action's expense (minerals and gas being the only resources to spend in StarCraft).

Longest Build Time and Shortest Build Time sort by the time required to finish a build action. However, because BOSS may need to wait for something else to finish before it can build a given action, the actual completion times may be longer than the build time. This is where the Latest Completion Time and Earliest Completion Time orderings come in, which account for such delays. These were all designed to minimize / maximize action duration, which may affect the total duration of build orders found early in the search.

The last ordering is Random, which shuffles the actions randomly at each step. This was added as another point of comparison besides ID-Increasing.

C. The DFBB Algorithm

The depth first branch and bound algorithm we use is described in Algorithm 1. In addition to search ordering, we also apply a landmark lower bound heuristic. This calculates a minimum amount of time it will take to reach the goal from each state and prunes the state if the total time exceeds the upper bound. The lower bound is calculated by finding the length of the longest chain of prerequisites (landmarks) that must be built in succession before the goal can be reached. This heuristic helps keep down computation times, especially for longer build orders.

Algorithm 1 Search Algorithm

```
function DFBB(startState, goal, ordering)
    bestBuildOrder = NaiveBuildOrder(startState, goal)
    bestMakespan = bestBuildOrder.makespan()
    stack = new Stack()
    stack.push(startState)
    while stack not empty do
        state = stack.pop()
        if state meets goal then
            makespan = state.finishTime()
            buildOrder = state.buildOrder()
            if makespan < bestMakespan then
                bestMakespan = makespan
                bestBuildOrder = buildOrder
            end if
        else
            for act in state.actions().sort(ordering).reverse() do
                next = state.copy()
                next.doAction(act)
                lowerBound = next.frame + Landmark(next, goal)
                if lowerBound < bestMakespan then
                    stack.push(next)
                end if
            end for
        end if
    end whilereturn bestBuildOrder
end function
```

D. Experiment Design

To test the various orderings, we ran multiple searches for each one. We chose ten goals for the search, which are as follows: four marines, one siege tank, one control tower, two firebats, one medic, four hydralisks, two mutalisks, one high templar, one carrier, and four zealots. These goals are from all three factions and were kept simple for the sake of reasonable computation times.

For each goal we ran DFBB with every search ordering. We measured the performance of each search by time elapsed, the number of search nodes expanded, and nodes per second. In the case of the Random ordering, we ran the search 50 times and took the average results.

III. Results

We found that search ordering gave significant improvements, though its effectiveness varied based on the ordering and the goal.

Table I shows the results in terms of time elapsed by the search. Each value has been divided by the time ID-Increasing elapsed searching for the same goal. This is to show improvement over the default and to facilitate comparisons across goals.

In general, the Naive Preferred F2 ordering and normal F2 ordering performed the best, coming first in terms of nodes expanded and time elapsed respectively. Our results demonstrate the utility of search ordering as a technique and the relative strengths of different methods in a StarCraft context.

The best orderings for time elapsed are F2, Random, and F3. F2 took, on average, 40% of the time ID-Increasing did. It was at worst 121% of ID-Increasing, for Marines x4, and at best 3%, for both High Templar x1 and Mutalisk x2. For Random the average was 43%, the worst was 133% (Hydralisk x4), and the best was 1% (Mutalisk x2). For F3, the average was 50%, the worst was 101%, and the best was 10%.

Importantly, the goals that these orderings worked best on were generally those that required longer build orders to complete, such as Mutalisk x2. This implies that savings may become more pronounced with more complex goals.

Measured by nodes expanded, which is the number of nodes the search visited and didn't prune, the results are slightly different. On average, the best orderings in terms of nodes expanded were Naive Preferred F2, Naive Preferred ID-Increasing, and F2. Naive Preferred F2 expanded 30% of the nodes of ID-Increasing on average. It had the worst performance for the goal of four Hydralisks, at 80%, and the best performance for the goal of two Mutalisks, at 4%. For Naive Preferred ID-Increasing the average was 32%, the worst was 87% (Hydralisk x4), and the best was 6% (Medic x1). For F2 the average was 43%, the worst 95% (Zealot x4), and the best 3% (Mutalisk x2).

Here we see the advantage of the Naive Preferred method, which results in more nodes being pruned from the search than orderings without it.

The third metric, nodes per second, helps explain the difference between the time elapsed and nodes expanded results. Nodes per second is equal to the nodes expanded divided by time elapsed. In general, the search orderings ran at similar nodes per second, with two exceptions: Naive Preferred ID-Increasing and Naive Preferred F2. These two were significantly slower processing each node, even if they resulted in less nodes needing to be searched. This ultimately resulted in them falling behind other metrics in time elapsed.

IV. Conclusion and Future Work

StarCraft build order optimization is a deceptively difficult problem. The game's tech tree, different types of build actions, and racial mechanics all serve to frustrate any simple approach to solving it. This very complexity is what makes StarCraft worth researching: it demands robust solutions to its problems.

In this research we have attempted to provide one such solution. We demonstrated the effectiveness of search ordering for StarCraft build order optimization using the Depth First Branch and Bound algorithm. We tested different orderings, across ten scenarios, and compared the results. On average the ordering that performed the best was F2, which sorts build actions based on how likely an action is to be chosen at a given time compared to how likely it is to be chosen at any other time. The Naive-Preferred modification, which moves the first build action in a naive build order to the front of the sorted list, also showed significant benefits. F2 performed best in terms of the real-time duration of the search, while Naive-Preferred F2 performed best in terms of the nodes expanded by the search.

The effectiveness of search ordering depends on the scenario. For simple goals, applying different orderings often fails

Time Elapsed / ID-Increasing	Marine x4	Siege Tank x1	Control Tower x1	Firebat x2	Medic x1	Hydralisk x4	Mutalisk x2	High Templar x1	Carrier x1	Zealot x4	Average
F2	1.21	0.09	0.06	0.09	0.27	0.89	0.03	0.03	0.38	0.96	0.40
Random (50 Samples)	0.85	0.45	0.33	0.28	0.22	1.33	0.01	0.02	0.26	0.54	0.43
F3	0.95	0.31	0.29	0.21	0.31	0.70	0.10	0.10	1.00	1.01	0.50
Naive Preferred F2	2.01	0.21	0.12	0.13	0.09	1.74	0.04	0.06	0.37	0.94	0.57
ID-Decreasing	1.00	0.45	0.33	0.10	0.07	1.98	0.52	0.50	0.35	0.47	0.58
Longest Build Time	0.97	0.48	0.37	0.11	0.07	2.00	0.53	0.50	0.35	0.48	0.59
Naive Preferred ID-Increasing	1.93	0.21	0.12	0.13	0.09	1.87	0.12	0.20	0.43	0.94	0.61
Greatest Mineral + Gas Cost	1.13	0.48	0.37	0.11	0.07	2.05	0.57	0.55	0.37	0.48	0.62
Least Prerequisites	0.96	0.47	0.35	0.15	0.11	1.99	0.52	0.50	0.45	0.84	0.63
Soonest Completion Time	1.18	0.30	0.27	0.95	1.06	0.97	0.18	0.18	0.36	1.06	0.65
Latest Completion Time	1.33	0.52	0.39	0.12	0.08	2.16	0.53	0.53	0.38	0.53	0.66
Least Mineral + Gas Cost	1.61	0.41	0.39	0.86	0.97	0.95	0.13	0.12	0.40	1.01	0.69
Shortest Build Time	1.11	0.42	0.39	0.86	0.97	0.89	0.18	0.17	1.00	1.00	0.70
Most Prerequisites	1.62	0.99	0.97	0.82	0.96	1.00	0.97	0.95	0.35	0.57	0.92
F4	0.99	1.00	0.98	1.00	1.01	1.01	0.94	0.93	0.98	1.01	0.98
F1	1.35	0.99	0.97	0.95	0.81	1.01	0.95	0.94	0.97	1.01	1.00
ID-Increasing	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

TABLE I: Time elapsed as a fraction of ID-Increasing's time elapsed. Green, white, and red respectively correspond to being shorter than, close to, and longer than ID-Increasing.

to produce meaningful benefits. For difficult goals however, the better orderings complete the search in a fraction of the time. Since difficult scenarios are precisely the ones that need efficient algorithms to solve, this is very convenient.

Search ordering as a technique, and the F2 ordering in particular, are thus successful in providing meaningful improvements in search time. Even so, the orderings that we have tested here are only a small sliver of those that are possible. New orderings can yet be devised to surpass those we have outlined.

We have also only scratched the surface in terms of using information from StarData to create build orders. Applying a machine learning approach may produce better search orderings than our method. Furthermore, a fundamental assumption of our work is that we are looking for optimal build orders. If that requirement is relaxed, and only high quality build orders are expected, it may be possible to find them using machine learning methods wholly different from heuristic search.

A definitive solution to StarCraft build order optimization remains elusive. Even so, the algorithm we have presented here represents a step towards achieving that goal and adds to the arsenal of techniques available to video game programmers and artificial intelligence researchers.

V. Acknowledgments

This research was conducted as part of Benjamin Stanley's honours thesis. He would like to thank Dr. Churchill, his supervisor, without whom this research would not have been possible.

Ben would also like to thank the Memorial University Department of Computer Science and particularly Cathy Hyde and Mark Hatcher for coordinating the honours program.

References

[1] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," *IEEE transactions on computational intelligence and AI in games.*, vol. 5, no. 4, pp. 293–311, 2013.

[2] D. Churchill and M. Buro, "Build Order Optimization in StarCraft," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 7, no. 1, pp. 14–19, Oct. 2011. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/12435

[3] Z. Lin, J. Gehring, V. Khalidov, and G. Synnaeve, "STARDATA: A StarCraft AI Research Dataset," 2017.

[4] G. Synnaeve, N. Nardelli, A. Auvolat, S. Chintala, T. Lacroix, Z. Lin, F. Richoux, and N. Usunier, "TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games," 2016.

[5] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1203–1212, 1989.