Search Ordering for StarCraft Build Order Optimization

by

Benjamin Stanley

Supervised by Dr. David Churchill

A thesis submitted to the

Department of Computer Science

to fulfill the requirements for the degree of

Bachelor of Science (Honour's).

Memorial University of Newfoundland

St. John's, NL, Canada

April 5, 2024

St. John's Newfoundland

Abstract

In real time strategy games, optimizing the order in which units and other items are built is essential to high level play. 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 apply the search ordering technique to this problem, along with the depth first search branch and bound algorithm. Search ordering guides the algorithm by sorting actions using some pre-defined ordering method. 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.

Acknowledgments

I would like to thank my supervisor, Dr. David Churchill, without whom this research would not have been possible.

I would also like to thank Memorial University of Newfoundland, the Faculty of Science, and the Department of Computer Science. In particular, I extend my gratitude to Cathy Hyde and Mark Hatcher for coordinating the honours program.

I am very grateful to all involved for granting me the opportunity to undertake this honours thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 StarCraft and Build Orders

StarCraft is a real-time strategy game developed by Blizzard Entertainment. In the game, players start off with a handful of units that they build up into an army with which to crush their opponents. Each player chooses one of three races to play as, each with unique units and mechanics. There are the Terrans, a human-like race that focus on mechanized warfare; the Protoss, an alien species who wield powerful psychic abilities; and the Zerg, a hive mind that overwhelms foes with their vast manpower. When devising a strategy, a player must consider the strengths and weaknesses of their race, the races 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 facilitated the game's popularity, even spawning a professional esports scene. It also attracted the attention of artificial intelligence researchers, who identified the game as a suitably complex environment to develop and test new models. Researchers make use of BWAPI (Brood War API)[1], an open source project that allows for running custom bots in StarCraft. 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 have 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. 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. In competitive play, build orders are used during a match's opening and determine a player's initial strategy[2]. There exist many well-defined build orders of this sort for the different race match-ups. Our research focuses on a different form of build order: a series of actions that result in the completion of a given goal. 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

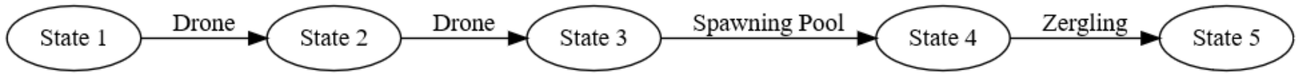

Figure 1.1: A common build order for the Zerg, represented as a graph.

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.

1.2 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. Heuristic search can be represented as graph traversal, and uses algorithms such as depth-first search, best-first search, and A* search.

Applying heuristic 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. If three such trees are constructed, each rooted on the start state for each StarCraft race, then all possible build orders and their resulting states would be represented. 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. These meth-

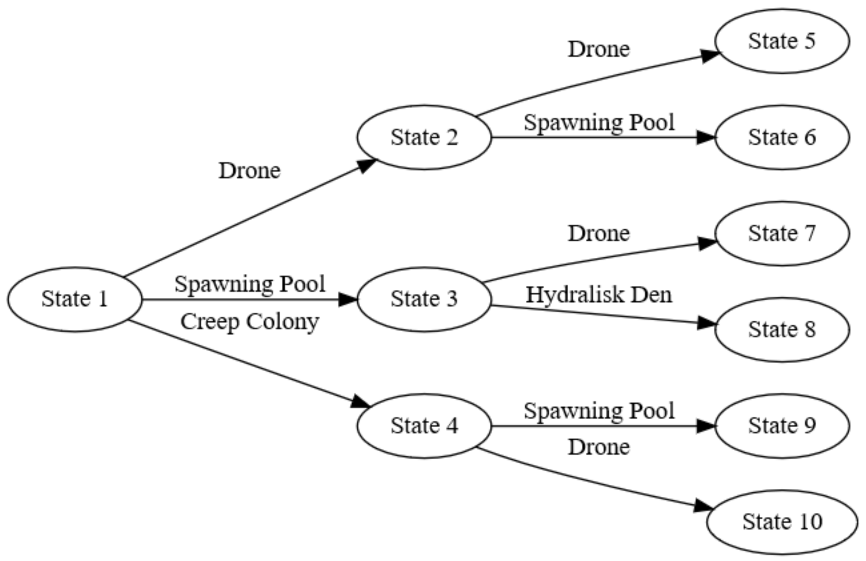

Figure 1.2: A tree of game states that represents some initial Zerg build orders.

ods restrict the size of the search space that actually needs to be considered, saving time. Heuristics are defined relative to the specific qualities of the problem domain, and we discuss the heuristics we use for StarCraft build order search in the methods chapter. The goal of our research is to develop a heuristic method to cut down the size of this search space as much as possible.

1.3 Search Ordering

Build order optimization is a subtype of the larger field of planning. New techniques developed for heuristic search over build orders will likely be applicable to other planning tasks. Furthermore, advancements in this area are directly useful in the field of video game development. Better algorithms for finding build orders allows for smarter AI opponents and faster games.

In pursuit of this goal, we introduce the search ordering technique, which sorts the build actions at each step of the search. When used with the depth first search branch and bound algorithm, this method alters the number of states that must be considered. We show that if certain criteria for ordering build actions are used, the search can be shortened to a fraction of its original duration.

Chapter 2

Related Work

StarCraft and its sequel, StarCraft 2, have been extensively researched as testing grounds for strategic AI development. Here we provide a look into some of that research, with a focus on build order optimization.

Research into StarCraft revolves around designing and improving upon bots that can play the game against each other or humans. For StarCraft, this is accomplished through BWAPI, the Brood War API, which facilitates running custom C++ bots[1]. 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, 3]. For a detailed review of these competitions and the history of early research in this field we refer the reader to [3], a survey of pre-2013 work.

Build order optimization is one of the many tasks bots must undertake to play StarCraft. Early techniques for solving this problem used means-end analysis (MEA) and were based not on StarCraft, but Wargus, an open source version of Warcraft 2[4]. This method was initially applied to build order optimization by Chan et al.[4], then improved upon by Branquinho and Lopes with their MeaPop and SLA* algorithms[5].

This paper follows research by Churchill and Buro[6] that deployed the depth first branch and bound algorithm for StarCraft build order optimization. To run this search they developed the Build Order Search System (BOSS), a simulator of StarCraft's build mechanics. We explain both of these in more detail in chapter 3.

Churchill, Turo, and Kelly[7] later used BOSS to address a variant of build order optimization where they maximize metrics representing firepower rather than search for a specific goal. They use Army Value (AV), the total resource cost of all military units constructed, as a proxy for firepower, optimizing for either AV itself or its integral over time. This method addresses the need for expert knowledge or the development of some kind of system to decide on goals, neither of which are ideal.

Researchers of build order optimization have also applied a wide variety of other methods to this problem.

Gmyrek, Antkiewicz, and Myszkowski approach build order optimization in StarCraft 2 using a genetic algorithm[8]. They conceive of build order optimization as a planning problem and introduce GAPS, the Genetic Algorithm for Planning and Scheduling, as a solution.

A genetic algorithm was also used by Köstler and Gmeiner for StarCraft 2. Rather than optimizing for arbitrary goals, they focus on two types of objectives: the "Rush", which produces as many units of a desired type as possible in a given time limit, and the "Tech-Push", which produces a unit of a given type as quickly as possible. To make their system useful for non-programmers, they even created a simulator with a GUI to run the searches.

Elnabarowy, Arroyo, and Wunsch II[9] propose a Monte Carlo tree search based method for StarCraft 2 build order optimization, with the intention of adding deep learning in the future.

A related problem that people have researched is build order prediction: determining which build order (and thus which strategy) an opponent is using from limited information. For example, Cho, Kim, and Cho constructed a model for strategy prediction by deriving decision trees from StarCraft replay data[10]. Khan, Hassan, and Sukthankar approach the same problem in StarCraft 2 with the use of a transformer-based architecture[11].

Collecting data for StarCraft is notably difficult. StarCraft game replays are available online from real players, but converting them to a useful form is time-consuming. Thus, researchers often use precompiled databases. One such database is MSC, which focuses on preparing macro-management StarCraft 2 data in a way amenable to machine learning[12], and was used by [11]. Another is StarData[13], which we use in this research and explain in greater detail in the following chapter.

Chapter 3

Methodology

The algorithm we are testing is a variation on depth first search with branch and bound. We improve upon the normal version of this algorithm by ordering the edges at each step of the search using various sorting methods. This causes the search space to change in size due to the properties of branch and bound, speeding up or slowing down the search.

Rather than have the algorithm run StarCraft directly, we use BOSS, a StarCraft build order simulator. We also make use of StarData, a dataset of StarCraft replays, to derive some of the ordering methods.

With this, we are able to compare the efficiency of multiple search ordering methods across multiple game scenarios.

3.1 BOSS

Our research is a continuation of previous work done by Dr. David Churchill and Dr. Michael Buro. They created a simulator of StarCraft's build system to be used for heuristic search, called the Build Order Search System (BOSS)[6]. Using this system, they developed a depth first search branch and bound algorithm that forms the basis of this research. We will describe BOSS in some detail, as it is fundamental to our work.

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. BOSS even abstracts out significant decisions related to building, such as where workers are allocated and where buildings are placed. Building placement is completely ignored and worker allocation is done automatically under predefined rules, such as assigning three workers to a gas extractor when built. The assumptions made here may result in subpar results in fringe cases, but they are usually valid in real play and significantly simplify the simulation. BOSS is thus easy to use: after setting up the start state for a given race, it requires only the names of the desired build actions to be taken as input. In this way it conforms exactly to the specification of build orders described previously, without exposing any other kinds of decisions.

Despite this, BOSS is highly complex. StarCraft has a number of qualities that make it difficult to simulate in a heuristic search context. There are the aforementioned implicit tech trees that must be tracked, as well as the race-specific mechanics. Each build action has an associated build time, with the resource cost being paid at the start of construction and the item only functioning after it is finished. This introduces the possibility that the number of actions available to a player may increase simply through the passage of time, as buildings that are prerequisites of other actions are completed. An even greater difficulty however, is that at any given moment multiple actions may be in progress at the same time. StarCraft, being a real time strategy game, operates without turns and allows the player to take whatever actions are available to them at any time, regardless of whether or not other actions are in progress. These complications make challenging even the determination of what actions can be taken to transition a game state in the search tree, the fundamental operation in heuristic search. To resolve this BOSS takes legal actions to mean not those actions available at the current time step, but all those actions that are or will be available at some point without taking another action. When an action is taken, the system fast-forwards to the exact frame that action becomes possible and then starts it. This allows algorithms to effectively search through BOSS states.

It is important to note that BOSS, in its current iteration, cannot handle upgrades. Upgrades are one of the types of build actions that exist in StarCraft, they generally make certain units stronger. As of yet BOSS cannot handle the multiple levels that upgrades can have, so they have been left out of the system. Our research thus does not account for upgrades.

3.2 TorchCraft and StarData

The other project we use in our research is StarData, a database of StarCraft replays compiled by researchers at Facebook[13]. 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 potential 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 game to analyse them, and also sometimes results in replays recorded on one version of the game being unreliable on others. Processing this data in an efficient manner is nearly impossible.

To remedy this, Facebook's researchers collected a set of 65646 replays 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[13].

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[14]. It is effectively a wrapper around BWAPI for ease of use in a machine learning context. 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.

3.3 Depth First Branch and Bound

The basic algorithm we use is depth first search with branch and bound. Depth first search is a graph traversal algorithm that searches out from a single vertex to check the whole graph. At each vertex, the algorithm checks that vertex, then goes to the next adjacent vertex not previously checked. These steps are repeated, backtracking whenever a vertex has no unchecked adjacent vertices.

In a StarCraft build order context, each vertex represents a single game state and each edge is a build action. Depth first search on these game states means beginning at the start state for a given race, then choosing build actions to transition to new states. If the search finds a state that meets the given goal, it backtracks and continues searching through every

Algorithm 1 Depth First Search algorithm for StarCraft build orders

```
function DFS(startState, goal)
    bestBuildOrder = NULL
    bestMakespan = $\infty$
    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 action in state.legalActions().reverse() do  ▷ This is reversed because the states are taken from the stack in reverse order
                newState = state.copy()
                newState.doAction(action)
                stack.push(newState)
            end for
        end if
    end whilereturn bestBuildOrder
end function
```

other possible state until it has checked them all and can be sure it has found the optimal solution. The problem is that a StarCraft build order has no real limit, so basic depth first search would be searching through an infinite number of states and would never finish. This is where branch and bound comes in. First, a naive build order (see below) is used to determine an initial upper bound on the makespan of the build order. Then at each step in the search, if the makespan of the current build order exceeds the upper bound, it backtracks. If the search encounters a valid solution with a shorter makespan than the current upper bound, the upper bound becomes that makespan. This bound makes the search space finite and dramatically speeds up the search, while maintaining optimality.

Algorithm 2 Adding branch and bound to search

```
17: newState = state.copy()
18: newState.doAction(action)
19: if newState.finishTime() < bestMakespan then
20:     stack.push(newState)
21: end if
```

Algorithm 3 Adding naive build order to search

```
 2: bestBuildOrder = NaiveBuildOrder(startState, goal)
 3: bestMakespan = bestBuildOrder.makespan()
```

3.4 Naive Build Order

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 significantly limits the search space. Naive build orders can also be used for search ordering, as we describe in the following section.

In general, the specific process by which naive build orders are constructed is unimportant. Any simple and fast method for finding a valid build order will suffice. The naive build order should be of reasonable quality, but wasting time trying to make it perfect is inadvisable: if that were simple the search would be unnecessary. 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 provides good quality build orders in a short time frame.

3.5 Landmark Lower Bound

Another technique this algorithm uses is landmark lower bound. Just as nodes in the search can be trimmed using an upper bound, lower bounds can also be used. A lower bound is a makespan such that, for a given state, it is impossible for the search from that state to yield any valid solutions shorter than that makespan. If such a lower bound can be found, and it is greater than the upper bound, then the given state cannot possibly reach the goal in less time than the upper bound, so further searching from that state is unnecessary. Thus, the state space can be pruned.

The actual process of finding lower bounds is difficult, however, especially for StarCraft. In StarCraft multiple build actions can progress simultaneously, meaning that simply taking the sum of the build times of the necessary actions will not work. A method that works for scenarios such as this is the landmark lower bound. A landmark lower bound is derived from a series of actions that must happen in sequence, with each only able to start after the previous is finished. These actions are the landmarks. Taking the sum of their lengths gives a minimum possible completion time, which can be used as a lower bound. In StarCraft, this means finding the longest chain of prerequisite actions from the goal back to the given state. If an action has a prerequisite, then it cannot begin construction until construction of that prerequisite is finished. A sequence of build actions where each is prerequisite to the next thus forms a landmark lower bound, as parallelization is impossible.

For example, take the Nydus Canal, a Zerg building. The Nydus Canal (600 frames) has as its prerequisite the Hive, the Hive (1800 frames) has the Queen's Nest as a prerequisite, and the Queen's Nest (900 frames) has the Lair (1500 frames) as its prerequisite. If the

Algorithm 4 Adding landmark lower bound to the search

```
17: newState = state.copy()
18: newState.doAction(action)
19: lowerBound = newState.currentFrame() + LandmarkLowerBound(newState, goal)
20: if lowerBound < bestMakespan then
21:     stack.push(newState)
22: end if
```

Algorithm 5 Adding ordering to search

```
16: for action in state.legalActions().sort(ordering).reverse() do
17:     newState = state.copy()
18:     newState.doAction(action)
19:     lowerBound = newState.currentFrame() + LandmarkLowerBound(newState, goal)
20:     if lowerBound < bestMakespan then
21:         stack.push(newState)
22:     end if
23: end for
```

search is at a state that has none of those, but could build the Lair, then at minimum it would take $600 + 1800 + 900 + 1500 = 4800$ frames to build a Nydus Canal. If, say, the current state is at frame 2000 and the upper bound is 6000, then $2000 + 4800 = 6800 > 6000$, therefore further searching from this state is unnecessary.

3.6 Search Ordering

An important consequence of using the branch and bound approach is that it matters which solutions are found first. If a solution with a shorter makespan is found earlier, the upper bound will lower, and more states will be pruned from the search afterward. This can be accomplished by altering the order in which adjacent vertices in depth first search are visited, which is usually arbitrary. Changing the search order in this way still guarantees an optimal solution, but it may shorten (or lengthen) the search time.

Algorithm 6 Final search algorithm

```
function DFS(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 action in state.legalActions().sort(ordering).reverse() do
                newState = state.copy()
                newState.doAction(action)
                lowerBound = newState.currentFrame() + LandmarkLowerBound(newState, goal)
                if lowerBound < bestMakespan then
                    stack.push(newState)
                end if
            end for
        end if
    end whilereturn bestBuildOrder
end function
```

In the context of StarCraft build orders, this means that the order of legal actions at each state affects the length of the search. By devising build action ordering methods based off various heuristics, it is possible to make better StarCraft build order search systems. That is exactly the purpose of our research, and we have devised numerous ordering methods that we tested and compared. We analyse the results of those tests in the following chapter, but we first explain each ordering here.

3.6.1 ID-Increasing and ID-Decreasing

By default, a search run using BOSS will iterate through actions in the order in which they are listed within the system. Each action has an associated integer ID in BOSS, this default order corresponds to a least-to-greatest ordering of those IDs, hence this ordering is called ID-Increasing. ID-Decreasing is the reverse of ID-Increasing, going from greatest-to-least. The IDs assigned to each action are arbitrary but not random. Higher IDs generally correspond with being farther along in the tech tree, and many of the lowest IDs are for simple units such as workers.

3.6.2 F1-4: StarData Derived Orderings

These four ordering schemes are derived from the StarData dataset. 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 obtained each action's type and the game state's current time, then ranks them from greatest to least.

3.6.3 Naive Preferred ID-Increasing and Naive Preferred F2

These sorting methods are alterations on other sorting methods that prefer the first action suggested by a naive build order. At each node in the search a naive build order is created and the action it suggests first is put first in the list. The rest of the legal actions at that node are then sorted based on the original ordering (ID-Increasing or F2). This requires significantly more calculations at each node than the other orderings. ID-Increasing was chosen as a baseline and F2 was chosen because it performed the best in early tests.

3.6.4 Most and Least Prerequisites

These methods rank the legal actions based on how many prerequisites they have. Not only an actions immediate prerequisites are considered, but all actions that precede it on the tech tree (in other words, its recursive prerequisites). The Most Prerequisites ordering prefers actions with a higher number of prerequisites while the Least Prerequisites ordering is the opposite. These orderings are meant to represent how advanced actions are in the tech tree.

3.6.5 Greatest and Least Mineral + Gas Cost

These orderings are based off the sum of an action's mineral cost and gas cost. They are effectively most and least expensive orderings.

3.6.6 Longest and Shortest Build Time

Each action has an associated build time: the number of frames it takes to build. The Longest and Shortest Build Time orderings thus rank based on how fast an action can be completed.

3.6.7 Latest and Earliest Completion Time

These orderings differ from Longest and Shortest Build Time due to BOSS's mechanics. An action in BOSS is legal when it can be built without taking any other actions, but that does not mean it is immediately ready. There may be a delay where the system must wait for resources to accumulate, or a building to finish construction, or for any other condition to be met. These orderings take that delay into account as well as the build time inherent to each action that the build time orderings are based on. The completion time orderings rank actions on when they can actually be finished from the current state.

3.6.8 Random

This ordering simply shuffles the legal actions randomly. It is included as another baseline, along with ID-Increasing, with which to compare the other orderings. We ran this ordering 50 times for each scenario and used the mean results.

3.7 Build Order Goals

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.

We kept these goals simple to make the searches completable in a reasonable time frame. Without further optimizations, especially those that sacrifice optimality, the search varies wildly in duration and is not practical for real time video game scenarios. We have elected here to keep a simpler form of the search algorithm so as to maintain optimality and better see the effects of different ordering methods.

Chapter 4

Results

For all but one of the build goals we tested, changing the build ordering from the default ID-Increasing method offered better results. These results varied greatly based on the ordering method used, and also based on the build goal.

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. These results demonstrate the utility of search ordering as a technique and the relative strengths of different methods in a StarCraft context.

In this chapter we break down the test results based on the three metrics: nodes expanded, time elapsed, and nodes per second. For nodes expanded and time elapsed, we take the results as a fraction of ID-Increasing's result for each goal. This phrases everything in terms of how much shorter it is than ID-Increasing, the default, and facilitates comparison across goals.

For each section we then analyse the implications of these results and discuss anomalies.

We also compare the results running with the landmark lower bound and without. The results we discuss in sections 4.1, 4.2, and 4.3 are from tests that used the landmark lower

Nodes Expanded / 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
Naive Preferred F2	0.75	0.15	0.10	0.08	0.06	0.80	0.04	0.23	0.27	0.49	0.30
Naive Preferred ID-Increasing	0.81	0.15	0.10	0.08	0.06	0.87	0.12	0.25	0.32	0.49	0.32
F2	0.94	0.12	0.08	0.10	0.29	0.90	0.03	0.51	0.40	0.95	0.43
Random (50 samples)	0.88	0.56	0.42	0.30	0.24	1.36	0.65	0.30	0.31	0.56	0.56
ID-Decreasing	0.76	0.57	0.43	0.12	0.08	2.05	0.52	0.33	0.43	0.50	0.58
F3	0.94	0.31	0.29	0.21	0.32	0.70	0.10	1.00	1.00	1.00	0.59
Latest Completion Time	0.76	0.62	0.49	0.12	0.09	2.08	0.53	0.33	0.43	0.50	0.59
Longest Build Time	0.76	0.62	0.49	0.12	0.09	2.08	0.53	0.33	0.43	0.50	0.59
Greatest Mineral + Gas Cost	0.76	0.62	0.49	0.12	0.09	2.12	0.57	0.34	0.45	0.50	0.60
Soonest Completion Time	1.00	0.33	0.29	0.86	0.98	0.92	0.18	0.45	0.37	0.95	0.63
Least Prerequisites	0.76	0.60	0.47	0.17	0.13	2.06	0.52	0.50	0.54	0.87	0.66
Least Mineral + Gas Cost	1.00	0.44	0.42	0.87	0.98	0.96	0.13	0.51	0.42	1.00	0.67
Shortest Build Time	1.00	0.44	0.42	0.87	0.98	0.90	0.18	1.00	1.00	1.00	0.78
Most Prerequisites	1.00	0.99	0.99	0.83	0.96	1.00	0.97	0.32	0.37	0.58	0.80
F1	0.99	0.98	0.99	0.94	0.80	0.99	0.95	0.97	0.97	1.00	0.96
F4	0.99	1.00	1.00	0.99	0.99	0.99	0.94	0.97	0.97	1.00	0.98
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 4.1: Nodes expanded as a fraction of ID-Increasing's nodes expanded. Green, white, and red respectively correspond to being shorter than, close to, and longer than ID-Increasing.

bound. We discuss the results of the tests that did not use it in section 4.4.

4.1 Nodes Expanded

Nodes expanded refers to the number of game states that had to be searched to find and confirm the optimal solution. We generally consider nodes expanded to be the most important metric, even though time elapsed is the one that is directly relevant to any use case. This is because nodes expanded is independent of the state of the computer on which it is run, which may affect time elapsed. Furthermore, given the exponential growth of the search space, orderings that prune more nodes should fare better with increasingly complex goals.

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).

These results demonstrate the effectiveness of Naive Preferred orderings in particular. Simply by using adding Naive Preferred to the ordering, ID-Increasing cut its nodes expanded to a third.

This also clearly identifies F2 as the superior underlying search ordering. It is the best of all orderings if Naive Preferred orderings are ignored, and its Naive Preferred version is best overall.

4.1.1 Hydralisks, Marines, and Zealots

All orderings were noticeably less effective for the goals Marine x4, Zealot x4, and Hydralisk x4. In contrast, the search orderings tended to do much better for Siege Tank x1, Control Tower x1, Mutalisk x2, High Templar x1, and Carrier x1. These build orders are comparatively longer and more complex, which implies that the effectiveness of search ordering as a technique increases with goal difficulty. This correlation is in line with the theory: in an exponentially increasing search space, increased node pruning would result in exponential savings. Therefore superior search orderings should increasingly perform better the more complex the goal, as these examples show.

There is however a complication, as neither Firebat x2 nor Medic x1 are meaningfully longer than Marine x4, Zealot x4, or Hydralisk x4. Furthermore, of those three goals, Hydralisk x4 had the longest build order, both in terms of time and number of actions, yet

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 4.2: 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.

it had the worst results. These irregularities indicate that build order length is not the only factor in search ordering performance. Further testing would be required to identify the specific variables at play, though with the high complexity of StarCraft's systems we suspect that might be difficult.

4.2 Time Elapsed

Time elapsed is simply the total time necessary for the search to finish. Like nodes expanded, it is a measure of the length of the search. However, unlike nodes expanded, time elapsed is susceptible to the conditions of the computer on which the search ran, and is thus less reliable. That being said, the time it takes the search to complete is ultimately what matters in any system that might be using this search, so it demands attention.

Unlike with nodes expanded, 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%.

The Naive Preferred orderings performed notably worse in time elapsed. As a result, F2 came out on top, closely followed by Random. Again this shows the superiority of the F2 ordering.

4.2.1 Random Ordering

An interesting result is the apparent strength of Random as an ordering. With a sample size of 50 it is possible that these results are not representative of the method's performance in general. However, assuming the results are somewhat accurate, this indicates that the simple act of randomizing the search is more effective than most of the other orderings that try to apply some logic to the search. We can only speculate as to the reasons why this might be the case. Given that all of these results are relative to ID-Increasing, it may be that ID-Increasing and almost all of the other orderings are actually quite bad. It may also be that there is some feature of the Random ordering that makes it a genuinely good ordering method. We suspect, for example, that Random's success can be attributed to its inconsistency. Random shuffles the build actions at every node, making it completely different each time. Most of the other orderings, by contrast, will put the actions in very similar or identical orders over all states. It may be possible that constantly trying different orders for the build actions is better than any set logic of which actions are better. Further research would be necessary to confirm this, however.

Nodes Per Second	Marine x4	Siege Tank x1	Control Tower x1	Firebat x2	Medic x1	Hydralisk x4	Mutalisk x2	High Templar x1	Carrier x1	Zealot x4	Average
Longest Build Time	128080	152803	145018	157744	195362	140016	124833	144968	126677	162290	147779
ID-Decreasing	124875	150881	145063	157791	195425	139809	124517	145116	126976	162665	147312
Random (50 Samples)	169068	148659	138757	148091	178732	137804	122174	142957	124583	161571	147240
Least Prerequisites	130722	152416	144414	153537	186221	139729	124786	140765	123723	161120	145743
Greatest Mineral + Gas Cost	110254	152178	144719	157946	194089	139665	124995	144472	126615	161488	145642
F2	127459	151647	145246	148418	171125	136041	122977	125053	111482	153326	139277
Latest Completion Time	93613	141495	135107	140997	176817	129525	118519	135743	117946	145609	133537
F3	162768	121710	110196	138194	168901	135787	123597	112323	103476	153525	133048
Shortest Build Time	147982	127574	117259	138258	162766	136570	124269	113077	103798	154762	132632
ID-Increasing	164569	119340	109196	137731	162173	134658	118746	113172	104012	155534	131913
F4	164961	119353	111322	136127	160162	132321	120230	112466	103412	152754	131311
Least Mineral + Gas Cost	101919	127981	117268	137987	162477	135870	124570	123530	108784	154368	129475
Most Prerequisites	101181	119970	111779	139233	162487	134210	121838	130909	109674	158647	128993
Soonest Completion Time	139976	130347	121055	125034	148891	127060	117837	121812	108458	140203	128067
F1	120039	118147	110729	135674	159631	132443	120245	112505	103630	153517	126656
Naive Preferred F2	61686	85626	85342	82218	101709	62137	82126	85259	76553	81863	80452
Naive Preferred ID-Increasing	68780	85417	85674	83013	101920	62456	70415	85504	76106	82100	80138

Table 4.3: Nodes per second. Green, white, and red respectively correspond to being faster than, close to, and slower than the mean nodes per second.

4.3 Nodes Per Second

Nodes per second is nodes expanded divided by time elapsed (in seconds). For nodes per second we do not present the data as a fraction of ID-Increasing's results, as it is already comparable across different goals. Compared to nodes expanded and time elapsed, there is little difference in nodes per second across orderings. There are however two exceptions to this: Naive Preferred ID-Increasing and Naive Preferred F2. Both of these orderings are significantly slower than all others, regardless of the goal.

The slow speed of the Naive Preferred orderings is not surprising. Each ordering generates a naive build order at every node to guide the search. Although that process is quite fast, it is still far slower than the other orderings, which are usually just looking up values and comparing them. This necessarily leads to processing fewer nodes per second.

The weakness of the Naive Preferred orderings in time elapsed compared to nodes expanded is a direct result of this slower speed. Naive Preferred may process less nodes than F2, for example, but the increase in the amount of time per node means that the total time is longer. So long as that is true, F2 remains the best ordering. However, if the correlation between the effectiveness of search orderings and the length of the build order holds, it is likely the Naive Preferred orderings will pull ahead for more complex goals. As the search space increases exponentially with build order length, the effect of savings in nodes expanded will become more and more pronounced. This implies that eventually the difference in nodes expanded between Naive Preferred and the other orderings will be so large that the slower speed will not matter. This would make Naive Preferred F2 the best ordering.

4.4 Without Landmark Lower Bound

We ran each test both with and without the landmark lower bound enabled. In general the landmark lower bound was beneficial, even essential. Even so, the results were not universally better with the lower bound, as we explore below.

The tables here are all formulated such that each cell is equal to $\frac{\text{value with lower bound}}{\text{value without lower bound}}$. The cells that say NULL are where the search could not finish in a reasonable amount of time without the landmark lower bound activated.

In terms of nodes expanded, the landmark lower bound consistently provides improvements. Just as there was less improvement from the search orderings for the x4 Marine, Hydralisk, and Zealot goals, there is less improvement from the landmark lower bound for those goals as well.

For time elapsed, the use of the landmark lower bound produced savings in line with that of nodes expanded, with the notable exception of Mutalisk x2, where the results varied wildly. We cannot begin to guess why the search was sometimes much faster without the

Nodes Expanded - Landmark Lower Bound Comparison	Marine x4	Siege Tank x1	Control Tower x1	Firebat x2	Medic x1	Hydralisk x4	Mutalisk x2	High Templar x1	Carrier x1	Zealot x4
ID-Increasing	0.89	0.16	0.12	0.28	0.18	0.63	0.08	0.07	NULL	0.98
ID-Decreasing	0.87	0.10	0.06	0.14	0.08	0.74	0.04	0.02	NULL	0.96
F1	0.89	0.16	0.13	0.28	0.16	0.63	0.08	0.06	NULL	0.98
F2	0.88	0.04	0.02	0.13	0.11	0.64	0.00	0.04	NULL	0.98
F3	0.88	0.10	0.06	0.23	0.12	0.59	0.02	0.07	NULL	0.98
F4	0.89	0.16	0.12	0.27	0.18	0.63	0.08	0.06	NULL	0.98
Naive Preferred Default	0.88	0.05	0.02	0.11	0.06	0.62	0.02	0.02	NULL	0.96
Naive Preferred F2	0.87	0.05	0.02	0.11	0.06	0.61	0.01	0.02	NULL	0.96
Most Prerequisites	0.89	0.16	0.12	0.24	0.18	0.63	0.08	0.03	NULL	0.97
Least Prerequisites	0.87	0.10	0.06	0.17	0.10	0.74	0.04	0.03	NULL	0.98
Greatest Mineral + Gas Cost	0.87	0.10	0.06	0.14	0.08	0.75	0.04	0.02	NULL	0.96
Least Mineral + Gas Cost	0.89	0.10	0.07	0.27	0.18	0.64	0.02	0.04	NULL	0.98
Longest Build Time	0.87	0.10	0.06	0.14	0.08	0.74	0.04	0.02	NULL	0.96
Shortest Build Time	0.89	0.10	0.07	0.27	0.18	0.63	0.02	0.07	NULL	0.98
Latest Completion Time	0.87	0.10	0.06	0.14	0.08	0.74	0.04	0.02	NULL	0.96
Soonest Completion Time	0.89	0.08	0.05	0.27	0.18	0.63	0.02	0.03	NULL	0.98
Random (50 samples)	0.91	0.10	0.06	0.20	0.11	0.70	NULL	0.02	NULL	0.97

Table 4.4: Nodes expanded with lower bound as a fraction of nodes expanded without lower bound. Green, white, and red respectively correspond to the search with the lower bound being shorter than, close to, and longer than the search without the lower bound.

landmark lower bound, despite searching far less nodes as shown in the previous table. Given that this is real runtime on a computer and is thus vulnerable to outside factors, it may be a fluke.

With nodes per second the disadvantage of using the landmark lower bound becomes clear: it requires significantly more computation at each node. Combined with the irregularities in time elapsed for the Mutalisk x2 goal, this suggests that the calculation of the landmark lower bound should be improved if possible. That being said, as the many NULL values attest to, the landmark lower bound should not be abandoned, even if it slows down the search in certain cases.

Time Elapsed - Landmark Lower Bound Comparison	Marine x4	Siege Tank x1	Control Tower x1	Firebat x2	Medic x1	Hydralisk x4	Mutalisk x2	High Templar x1	Carrier x1	Zealot x4
ID-Increasing	2.19	0.32	0.25	0.57	0.34	1.25	20.05	1.61	NULL	1.82
ID-Decreasing	2.90	0.17	0.09	0.28	0.13	1.63	9.57	0.93	NULL	1.98
F1	2.30	0.33	0.25	0.57	0.30	1.41	19.37	1.50	NULL	1.95
F2	2.12	0.08	0.03	0.26	0.20	1.42	1.22	0.05	NULL	1.93
F3	1.24	0.21	0.13	0.50	0.22	1.27	4.32	0.16	NULL	1.95
F4	1.42	0.32	0.24	0.56	0.34	1.41	19.00	1.49	NULL	1.95
Naive Preferred Default	1.36	0.06	0.02	0.14	0.07	0.79	1.40	0.18	NULL	1.27
Naive Preferred F2	1.50	0.06	0.02	0.14	0.07	0.78	0.68	0.06	NULL	1.27
Most Prerequisites	3.45	0.32	0.24	0.50	0.33	1.44	19.51	2.03	NULL	1.96
Least Prerequisites	2.75	0.18	0.10	0.35	0.19	1.64	9.58	0.78	NULL	2.00
Greatest Mineral + Gas Cost	3.24	0.18	0.10	0.28	0.14	1.65	10.09	1.02	NULL	2.01
Least Mineral + Gas Cost	3.51	0.20	0.13	0.57	0.32	1.43	3.97	0.21	NULL	1.98
Longest Build Time	2.12	0.18	0.10	0.29	0.14	1.64	9.59	0.93	NULL	1.97
Shortest Build Time	1.46	0.20	0.13	0.57	0.33	1.41	5.24	0.27	NULL	1.97
Latest Completion Time	2.25	0.17	0.10	0.26	0.13	1.50	8.89	0.90	NULL	1.76
Soonest Completion Time	1.59	0.13	0.09	0.53	0.32	1.29	4.91	0.21	NULL	1.77
Random (50 samples)	2.15	0.18	0.10	0.43	0.20	1.55	NULL	0.04	NULL	2.07

Table 4.5: Time elapsed with lower bound as a fraction of time elapsed without lower bound. Green, white, and red respectively correspond to the search with the lower bound being shorter than, close to, and longer than the search without the lower bound.

Nodes Per Second - Landmark Lower Bound Comparison	Marine x4	Siege Tank x1	Control Tower x1	Firebat x2	Medic x1	Hydralisk x4	Mutalisk x2	High Templar x1	Carrier x1	Zealot x4
ID-Increasing	0.41	0.50	0.49	0.48	0.53	0.50	0.49	0.52	NULL	0.54
ID-Decreasing	0.30	0.57	0.61	0.48	0.57	0.45	0.51	0.65	NULL	0.49
F1	0.39	0.50	0.51	0.49	0.53	0.45	0.51	0.53	NULL	0.50
F2	0.42	0.57	0.60	0.50	0.54	0.45	0.49	0.58	NULL	0.51
F3	0.71	0.48	0.47	0.46	0.54	0.46	0.50	0.53	NULL	0.50
F4	0.62	0.50	0.51	0.49	0.53	0.45	0.51	0.53	NULL	0.50
Naive Preferred Default	0.64	0.88	0.92	0.82	0.91	0.79	0.79	0.92	NULL	0.76
Naive Preferred F2	0.58	0.89	0.91	0.82	0.89	0.78	0.89	0.92	NULL	0.76
Most Prerequisites	0.26	0.50	0.50	0.49	0.53	0.44	0.51	0.59	NULL	0.49
Least Prerequisites	0.32	0.57	0.60	0.48	0.54	0.45	0.51	0.63	NULL	0.49
Greatest Mineral + Gas Cost	0.27	0.57	0.61	0.49	0.57	0.45	0.51	0.64	NULL	0.48
Least Mineral + Gas Cost	0.25	0.50	0.51	0.48	0.55	0.45	0.50	0.56	NULL	0.50
Longest Build Time	0.41	0.58	0.61	0.49	0.57	0.45	0.51	0.65	NULL	0.49
Shortest Build Time	0.61	0.50	0.51	0.48	0.53	0.45	0.50	0.52	NULL	0.50
Latest Completion Time	0.39	0.61	0.62	0.55	0.60	0.50	0.52	0.66	NULL	0.55
Soonest Completion Time	0.56	0.57	0.57	0.52	0.56	0.49	0.52	0.88	NULL	0.55
Random (50 samples)	0.43	0.57	0.60	0.48	0.55	0.45	NULL	0.62	NULL	0.47

Table 4.6: Nodes per second with lower bound as a fraction of nodes per second without lower bound. Green, white, and red respectively correspond to the search with the lower bound being faster than, close to, and slower than the search without the lower bound.

Chapter 5

Conclusion

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 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. Furthermore, testing a larger set of goals could answer some of the questions we have posed in terms of how these ordering methods function.

In particular, it may be interesting to see if randomized sorting is consistently beneficial. If that were found to be true for depth first branch and bound algorithms in general, it would provide a simple, widely applicable heuristic to speed up searches.

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.

Bibliography

[1] BWAPI. The Brood War API, 2017. `https://bwapi.github.io/`.

[2] Liquipedia. Build order, 2023. `https://liquipedia.net/starcraft/Build_order`.

[3] Santiago Ontanon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE transactions on computational intelligence and AI in games.*, 5(4):293–311, 2013.

[4] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Online Planning for Resource Production in Real-Time Strategy Games. In *Proceedings of the Seventeenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'07, page 65–72. AAAI Press, 2007.

[5] Augusto A. B. Branquinho and Carlos R. Lopes. Planning for resource production in real-time strategy games based on partial order planning, search and learning. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 4205–4211, 2010.

[6] David Churchill and Michael Buro. Build Order Optimization in StarCraft. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 7(1):14–19, Oct. 2011.

[7] David Churchill, Michael Buro, and Richard Kelly. Robust Continuous Build-Order Optimization in StarCraft. *IEEE Conference on Games*, 2019.

[8] Konrad Gmyrek, Michał Antkiewicz, and Paweł B. Myszkowski. Genetic Algorithm for Planning and Scheduling Problem – StarCraft II Build Order Case Study. In *2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 131–140, 2023.

[9] Islam Elnabarawy, Kristijana Arroyo, and Donald C. Wunsch II au2. StarCraft II Build Order Optimization using Deep Reinforcement Learning and Monte-Carlo Tree Search, 2020.

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

[11] Muhammad Junaid Khan, Shah Hassan, and Gita Sukthankar. Leveraging Transformers for StarCraft Macromanagement Prediction. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1229–1234, 2021.

[12] Huikai Wu, Yanqi Zong, Junge Zhang, and Kaiqi Huang. MSC: A Dataset for Macro-Management in StarCraft II, 2023.

[13] Zeming Lin, Jonas Gehring, Vasil Khalidov, and Gabriel Synnaeve. STARDATA: A StarCraft AI Research Dataset, 2017.

[14] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games, 2016.