Partial Synthesis of Heuristic Search Algorithms

by

Matthew Gallivan

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Heuristic search is a core area of Artificial Intelligence (AI) with numerous applications. In video games it is commonly used to calculate paths of AI-controlled agents. Traditionally, heuristic search algorithms have been designed by humans. Recent work attempted to synthesise heuristic search algorithms by automatically combining elements of published algorithms. In doing so, researchers defined a synthesis space for heuristic search algorithms and then automatically searched through that space. We extend this line of work and make the following contributions. First, we define a richer space of algorithms using a finer set of building blocks. This space is constructed using a context-free grammar. We then show that in the new space we can automatically synthesise higher performing real-time heuristic search algorithms. We evaluate these algorithms over benchmark pathfinding problems taken from video games and show that our synthesis method outperforms existing work.

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

—T. S. Eliot, *Little Gidding*

Acknowledgements

Throughout my degree I have received tremendous support. My supervisor, Dr. Vadim Bulitko, has made this thesis possible. I am a better researcher for having been mentored by him. He has guided me over every hurdle of this journey and he has, time and again, proven to be knowledgeable, patient and generous. The research contained herein is the result of a collaborative process between myself and Dr. Bulitko. Chapters 4 and 6 are an adaptation of work from an unpublished paper that was co-written by both of us. The term we is used throughout to reflect these facts.

I thank Dr. Michael Buro and Dr. Levi H. S. Lelis for serving on my examining committee, and Dr. Sarah Nadi for acting as my examination chair. I would also like to thank Dr. Matthew R. G. Brown for his advice on the statistical aspects of my research. I thank Dr. Dana Cobzas, Trevor Graham and Dr. Mehrnoush Mirhosseini for their immensurate backing and encouragement.

I would like to thank Lana Cook for her unfaltering support in every meaning of the word. She has read every word I have written over the past two years and has spent innumerable hours discussing the finer points of heuristic updates.

Finally, I would like to thank my mother, Valerie Gallivan; my father, Dennis Gallivan; and my sister, Andrea Gallivan. They have been the scaffolding of my life for over three decades and where I stand now owes completely to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Non-playable characters (NPCs) are computer-controlled agents that operate in the environment of a video game. Many NPCs navigate within an environment containing walls, human-controlled player characters and other NPCs. To appear responsive and maintain realism these NPCs must find their routes, or paths, quickly. The task of moving from a start location to a goal location in a complex environment is known as *pathfinding* (Figure 1.1).

1.1 Motivation

There are a number of algorithms capable of solving pathfinding tasks. The A* algorithm (Hart et al., 1968) finds a shortest possible path from start to goal. However, a drawback of A* is that it must compute the entire path before the agent is able to move. For games that have a number of computationally intensive processes running simultaneously, computing entire paths for each agent can be infeasible. Furthermore, calculating the optimal path is often unnecessary; it serves to find a realistic-looking path for an agent.

While some games have used hierarchical pathfinding (Sturtevant & Geisberger, 2010) to improve computation time, another approach is to place a constraint on the heuristic search, upper-bounding its per-move planning time by a constant that is independent of the size of the environment.

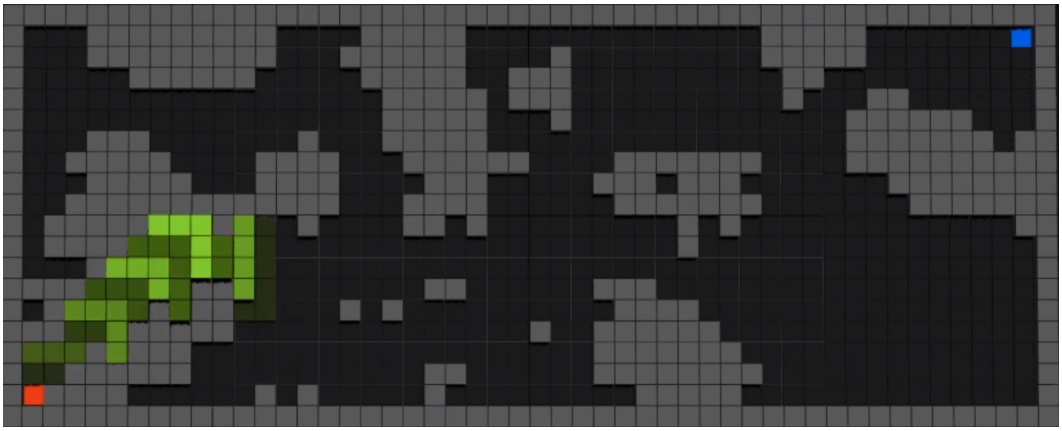

Figure 1.1: A pathfinding task displayed in Path Finding Visualizer (Bodyul, 2010). Red is the start state, blue is the goal state, light grey cells are impassable and green displays the heuristic.

Algorithms that satisfy this constraint are known as *real-time heuristic search* (RTHS) algorithms (Korf, 1990). These algorithms forego guaranteed optimality and, in exchange, are able to return partial suboptimal paths; these partial paths allow agents to move before a full path has been calculated. The seminal Learning Real-time A* (LRTA*) algorithm (Korf, 1990) accomplished this by interleaving planning, acting and learning. The ability to quickly produce partial paths makes RTHS well-positioned for video game pathfinding.

However, RTHS algorithms can suffer from extensive state revisitation (also known as scrubbing) (Sturtevant & Bulitko, 2014, 2016). Over the last several decades RTHS researchers have addressed scrubbing in a number of different ways, including using different learning and action-selection rules (Bulitko & Sampley, 2016; Hernández & Baier, 2012; Rivera et al., 2015) and through the removal of specific graph vertices (Hernández et al., 2017; Sharon et al., 2013). While each of these methods has led to better performance over the original LRTA* algorithm, problems still exist. For example, certain methods contain hyperparameters which must be tuned on a per-problem basis. Furthermore, combining such methods can lead to their interacting in unexpected ways.

Thus, there exists the question of how best to combine such methods in a high-performing RTHS algorithm.

Recent work (Bulitko, 2016a, 2016b) has addressed this by conducting a search through a parameterised space of RTHS algorithms. More generally, automatic generation of algorithms, known as *program synthesis* (Manna & Waldinger, 1971), offers several advantages over traditional manual algorithm design. First, it removes the very substantial and typically unreported human time and effort needed to design a new algorithm for a particular domain. Second, it means that algorithms can be synthesised on a per-environment or even per-problem basis, allowing them to exploit latent features within those domains. Third, insights gained from synthesised algorithms can be used to improve the design of the algorithm space in an iterative human-machine design process. Finally, allowing AI to configure itself to the problem at hand is a valuable step towards artificial general intelligence.

A crucial design choice when using program synthesis is determining the scope and complexity of the underlying space of algorithms. A simple space might consist of a single tunable hyperparameter. A space with more representational power might be defined as all valid programs in the C language. The former is easier to search, although unlikely to contain algorithms that are as highly performant as those found in the latter. A desirable space has enough representational power to contain a high-performance algorithm but is small enough that the synthesis is computationally feasible. This motivates our work extending the space of heuristic search algorithms and evaluating the resulting synthesised algorithms in real-time video game settings.

1.2 Contributions

This thesis makes the following contribution:

- We propose a new space of RTHS algorithms; extending an important subset of a previously published space. Our new space is defined by a context-free grammar. This enriches the representation of synthesisable heuristic search algorithms while retaining the ability to represent a number of published algorithms.

- We then evaluate the new space by conducting a program synthesis over it. Candidate algorithms synthesised from the new space of algorithms outperform candidate algorithms synthesised from the space published in previous work when evaluated for pathfinding on a set of video game maps.

1.3 Organisation

This thesis is organised as follows: Chapter 2 mathematically formulates the notion of a search problem and program synthesis. Chapter 3 reviews work related to the problem at hand. Chapter 4 presents our approach. Chapter 5 analyses our algorithmic space. Chapter 6 evaluates our new approach empirically. Chapter 7 proposes future work. Chapter 8 concludes with a summary.

Chapter 2

Problem Formulation

In this chapter, we define the problem of synthesising heuristic search algorithms. In Section 2.1 we define a heuristic search problem. Since we evaluate our methods in real-time settings, we introduce the notion of a real-time constraint on heuristic search in Section 2.1.1. In Section 2.2 we define our performance measure. Finally, in Section 2.3 we frame the task of finding high-performance algorithms as an optimisation problem.

2.1 Background

We adopt the definition of a search problem from the literature (Bulitko, 2016a). Given our focus on NPCs we present search in an agent-centered manner. A *heuristic search problem* is defined as the tuple (S, E, c, s_0, s_g, h). S is a finite set of states and $E \subset S \times S$ is a finite set of undirected edges connecting those states. Together S and E represent a *search graph*. The cost to traverse an edge from one state to another is given by the function $c : E \to \mathbb{R}^+$. The start state is $s_0 \in S$ and the goal state is $s_g \in S$. A *search agent* begins in state s_0. The *neighbourhood* of a state is defined by the function $N(s) = \{s_n \in S : (s, s_n) \in E\}$. A state is *expanded* when all of its immediately neighbouring states are considered. The total number of states expanded by an agent a on a problem p is denoted by $\mathcal{E}(a, p)$.

A search problem is solved when an agent finds a *solution path* $\psi = (s_0, s_1, \ldots, s_k)$ where $s_k = s_g$. The cost of this path is the sum of all edge costs between its consecutive states: $C(a, p) = \sum_{i=0}^{k-1} c(s_i, s_{i+1})$. The *optimal path* is a path with the lowest possible path cost for a given problem, represented by $C^*(p)$. The *suboptimality* of a path is the ratio between the path cost and the optimal path cost:

$$\alpha(a, p) = \frac{C(a, p)}{C^*(p)} \tag{2.1}$$

The best possible suboptimality is 1. A suboptimality of 3 indicates a path three times longer than optimal. In this work we assume all search problems to be solvable.

A search agent is also given a *heuristic function* $h : S \times S \to \mathbb{R}^+$. This heuristic serves as an estimate of the path cost between any two states, with the optimal costs between two states being represented by the function h^*. We shorten $h(s, s_g)$ to $h(s)$ when the goal state is clear from the context.

2.1.1 Real-time Heuristic Search

In *agent-centered search* (Koenig, 2001) the agent occupies a single state s_t at time t. Time advances in discrete steps $t \in \{0, 1, \ldots\}$. An agent can only occupy a single state at any time. If an agent is in the state s_t then at $t + 1$ it must move to a neighbour such that $s_{t+1} \in N(s_t)$. An agent is said to be *scrubbing* if it revisits any state it has already visited (Sturtevant & Bulitko, 2014, 2016). The agent is able to update its heuristic at each time step. The heuristic at time t is represented by h_t. The initial heuristic is $h_0 = h$.

A search agent is *real-time* if its planning time per move is bounded by a constant that is independent of the total number of states in the search graph (Korf, 1990). An algorithm is *complete* if it reaches s_g at some point in time (when the search problem is solvable). If, whilst solving a problem p,

Algorithm 1: LRTA* with cutoff

Input: problem $p = (S, E, c, s_0, s_g, h)$, max travel $C_{\max}$
Output: path $\psi = (s_0, s_1, \ldots, s_t)$

1 $t \leftarrow 0$
2 $s_t \leftarrow s_0$
3 $h_t \leftarrow h$
4 $\psi \leftarrow (s_0)$
5 **while** $s_t \neq s_g$ and $C(\psi) \leq C_{\max}$ **do**
6 $\quad s_{t+1} \leftarrow \arg\min_{s \in N(s_t)}(c(s_t, s) + h_t(s))$
7 $\quad h_{t+1}(s_t) \leftarrow \min_{s \in N(s_t)}(c(s_t, s) + h_t(s))$
8 $\quad \psi \leftarrow \text{append}(\psi, s_{t+1})$
9 $\quad t \leftarrow t + 1$

an agent's path exceeds a suboptimality of $\alpha_{\max} \cdot C^*(p)$, the agent is stopped and its suboptimality is recorded. Calculating this threshold requires precomputing the optimal cost $C^*(p)$.

2.1.2 Heuristic Search Algorithms

The Learning Real-time A* (LRTA*) algorithm (Korf, 1990) solves real-time heuristic search problems. At each time step it selects a neighbouring state, updates the current state's heuristic and moves to its next state. We augment the standard LRTA* algorithm with a cutoff, as can be seen in Algorithm 1. LRTA* consists of planning (line 6), learning (line 7) and movement (line 8) rules. It receives as input a search problem p and a maximum travel cost $C_{\max}$. On lines 1 through 4, we initialise the time, the current state, the current heuristic and the path ψ. Beginning on line 5, the algorithm continues pathfinding until it has found its goal state or reached its maximum travel cost. On line 6, it selects its neighbour with the lowest f score. In LRTA* we define the f score as $c(s_t, s) + h_t(s)$. On line 7, it updates the heuristic of the current state. On line 8 it adds to the path. Finally, on line 9, it updates the time step. If $C(\psi) > C_{\max}$ the path ψ may not end with the goal state.

2.2 Performance Measures

We define the *performance* of a real-time heuristic search algorithm evaluated over a set of problems $P = \{p_1, \ldots, p_n\}$ as its mean suboptimality over all problems in P:

$$\alpha(a, P) = \frac{1}{n} \sum_{i=1}^{n} \alpha(a, p_i) \tag{2.2}$$

The *solution time* is defined as the cumulative time to solve all search problems within the set P.

2.3 Program Synthesis

We denote the space of all possible heuristic search algorithms as A. The problem we tackle in this work is to find the optimal algorithm $a^* \in A$ with the lowest suboptimality over a set of problems $P = \{p_1, \ldots, p_n\}$. This can be viewed as the unconstrained optimisation problem (Boyd & Vandenberghe, 2014):

$$\underset{a \in A}{\text{minimize}}\ \alpha(a, P) \tag{2.3}$$

Our synthesis methods solve this optimisation problem by approximating a^*. Additionally we impose the following preferences. We seek a heuristic search algorithm with low suboptimality synthesised using little optimisation effort. We define optimisation effort (or *synthesis cost*) in terms of the number of state expansions performed during synthesis.

We also prefer synthesised algorithms that retain their performance when used on novel environments not seen during synthesis. Given an algorithm synthesised from a set of problems P_A, we consider it *portable* if its suboptimality over another set of problems P_b is less than or equal to the suboptimality of an algorithm synthesised and evaluated on P_b directly. Portable algorithms are beneficial as (i) they eliminate the need to run

synthesis for each environment and (ii) they are more likely to perform well in environments that dynamically change during gameplay.

Finally, we prefer synthesised search algorithms to be *compact* in their representation and *explainable* to human researchers. Such algorithms are easier to communicate to video game developers, provide insight and are more likely to be trusted. Thus, compact and explainable search algorithms are more likely to be deployed in a video game.

Chapter 3

Related Work

In this chapter we review existing work related to the problem of synthesising heuristic search algorithms, as defined in Chapter 2.

A number of extensions have been made to the LRTA* algorithm (Korf, 1990) since its inception. These include refinements to the planning, learning and movement rules of the LRTA* framework. For example, planning has benefited from extension to the selection of nodes expanded as found by Koenig and Sun (2009). Learning has been improved by adapting the heuristic to the planning step (Bulitko, 2004) and allowing updates to the heuristic over a number of states at each time step (Hernández & Meseguer, 2005). An improvement in the movement phase can be found in the work on depression avoidance (Hernández & Baier, 2012). The development of these extensions can be viewed as a manual search through a space of algorithms. While the algorithms developed are often highly-performant, the time to develop is on the order of months or years.

More recent work (Bulitko, 2016a, 2016b) synthesised real-time heuristic search algorithms over a space of hyperparameters. Each hyperparameter controlled an individual building block that affected the search of the algorithm. For example, the weight hyperparameter multiplied updates to the heuristic learning rule. The algorithms synthesised, both by using a

simple grid-based search as well as by using an evolutionary search, obtained lower suboptimalities than existing hand-crafted algorithms. Furthermore, during the search they scrubbed less on average. While the results were promising, the space of possible algorithms remained quite constrained, represented only by a vector of parameters controlling existing building blocks. We postulate that by expanding this existing space, a synthesis process will find even better algorithms.

Related to searching a space of algorithm parameters is searching a space of heuristic functions. Bulitko (2020) used an evolutionary algorithm to search through a space of heuristics defined by a grammar of algebraic expressions. This approach has been further improved by re-adding synthesised heuristics to be used in a new synthesis (Bulitko et al., 2021; Hernandez & Bulitko, 2021). Since the behaviour of heuristic search algorithms is determined in part by their heuristic, the synthesis of a high quality heuristic is crucial to the performance of the algorithm. This research runs parallel to our own — a future joint synthesis of algorithms and heuristics may produce even higher performing algorithms.

Machine learning has also been applied to the process of program synthesis. Muñoz et al. (2018) used a neural network in place of a movement rule; the network received a number of inputs concerning an agent's neighbouring states and output the agent's next move. The process of training the network can be framed as program synthesis if we consider each individual set of weights as a unique algorithm. The supervised learning is then a search over the space of algorithms representable by the network. The resulting networks beat or were comparable in suboptimality to some existing real-time heuristic search algorithms. However, an issue concerning this work and neural networks in general is that they can be difficult for humans to explain and communicate, which is one of our preferences.

In the domain of non-real-time heuristic search, Chen and Sturtevant (2019) examined properties of priority functions used by A* to shape its search. The priority function can affect the number of states expanded while A* finds an optimal solution. Since the priority functions are parameterised we can imagine forming a searchable space over their parameter set.

Search-based program synthesis (Gulwani et al., 2017) has emerged as an effective means of generating algorithms. A form of program synthesis, programming-by-example, constructs programs using input-output pairs. This is related to our use of heuristic search problems to synthesise algorithms, although we do not explicitly provide such pairs. Rather, we treat the suboptimality of an algorithm as the objective function to optimise.

The synthesis of video game pathfinding algorithms can also be viewed as procedural content generation (PCG) which has previously been applied to games, automatically creating quests, levels and sounds, among other content (Hendrikx et al., 2013). Our approach can be thought of as the generation of NPC pathfinding behaviour; although, we are unaware of any existing PCG work applied to real-time heuristic search algorithms.

Chapter 4

Our Approach

In this chapter we present our approach to synthesising real-time heuristic search algorithms. In Section 4.1 we describe the space of algorithms over which we will optimise. In Section 4.2 we describe how the suboptimality of a heuristic search algorithm is evaluated. Finally, in Section 4.3 we combine both the space of algorithms and their suboptimalities to craft a search method that synthesises programs.

4.1 Space of Algorithms

We begin by defining A, the space of algorithms. An ideal space would contain high-performing search algorithms and nothing else. Since we do not know what such algorithms are before they are synthesised, we instead define the space as an extension of a critical subset of a published space of RTHS algorithms.

4.1.1 Background

Hand-crafted extensions to the planning, learning and moving rules in LRTA* have been developed for decades. Recent research (Bulitko, 2016a, 2016b) has grouped a number of these extensions into a singular parameterised framework, which we refer to as parameterised LRTA*

Algorithm 2: Parameterised LRTA*

Input: problem $p = (S, E, c, s_0, s_g, h)$, parameters w, b, *lop*, *bt*, *da*
Output: path $\psi = (s_0, s_1, \ldots, s_g)$

1 $t \leftarrow 0$
2 $\psi \leftarrow (s_0)$
3 **while** $s_t \neq s_g$ **do**
4 **if** *da* **then**
5 $\mu \leftarrow \min_{s \in N(s_t)} |h_0(s) - h_t(s)|$
6 $\widetilde{N}(s_t) \leftarrow \{s \in N(s_t) : |h_0(s) - h_t(s)| = \mu\}$
7 **else**
8 $\widetilde{N}(s_t) \leftarrow N(s_t)$
9 $h_{t+1}(s_t) \leftarrow w \cdot lop_{s \in \widetilde{N}_b^{\mathrm{f}}(s_t)}(c(s_t, s) + h_t(s))$
10 **if** *bt* & $h_{t+1}(s_t) > h_t(s_t)$ & $t > 0$ **then**
11 $s_{t+1} \leftarrow s_{t-1}$
12 **else**
13 $s_{t+1} \leftarrow \arg\min_{s \in N(s_t)} c(s_t, s) + h_t(s)$
14 $\psi \leftarrow \mathrm{append}(\psi, s_{t+1})$
15 $t \leftarrow t + 1$

(Algorithm 2)[1]. This has served as the space of algorithms A in existing work. It receives as input a search problem p along with the following parameters: w which multiplies the heuristic update, b which restricts the available neighbours as sorted by their f score, *lop* which determines the learning operator used in the heuristic update, and *bt* and *da* which toggle the backtracking and depression avoidance components, respectively. Line 1 initialises the time, line 2 initialises the path and on line 3 the search commences until the current state s_t is the goal state. If depression avoidance has been enabled, lines 4 through 6 restrict the neighbourhood to only those neighbours that have received the minimal heuristic update. Line 9 updates the heuristic of the current state using the weight parameter w and the learning parameter *lop*, along with a neighbourhood restriction as

[1] We exclude expendable states in our re-creation of the algorithm. In the four-neighbour maps seen in Chapter 6, the expendable states (Sharon et al., 2013) building block does not cull any neighbours and is thus superfluous to our analysis.

dictated by b. The parameter $b \in [1, 4]$ indicates the number of neighbours to include in $\widetilde{N}$. For example, if $b = 2$ only the two neighbours with the lowest f score are stored in $\widetilde{N}$. On lines 10 and 11, if backtracking has been enabled using the bt parameter and the newly calculated heuristic is greater than the previous heuristic of the current state, the agent moves to s_{t-1}. Otherwise, on line 13, the agent moves to the neighbour with the lowest f score. It uses the entire neighbourhood N for this step. On line 14, we append s_{t+1} to our path. Finally, on line 15, the time is incremented and the loop continues.

4.1.2 Learning Rules in a Context-Free Grammar

Our hypothesis is that by extending an important part of the space of algorithms defined by parameterised LRTA* we will be able to represent and synthesise heuristic search algorithms with lower suboptimalities. Parameterised LRTA* has a number of numeric and boolean parameters. Replacing the learning rule in parameterised LRTA* with a richer representation allows us to represent additional algorithms.

We choose to replace the learning rule of parameterised LRTA* with one represented by a *context-free grammar*. This decision is based on promising results synthesising heuristics using context-free grammars for real-time search (Bulitko, 2020; Bulitko et al., 2021). A grammar is a specification of how to construct strings in a language, given a set of tokens and production rules. Each token can be nonterminal, meaning it can be replaced using a production rule, or terminal, an elementary symbol with no associated production rule. A context-free grammar (CFG) requires that the left-hand side of any rule is a single nonterminal token. Our context-free grammar is

below[2]:

$$
\begin{aligned}
T &\leftarrow O \mid C \mid V \\
O &\leftarrow T + T \mid T \times T \\
O &\leftarrow \min\{T, T\} \mid \max\{T, T\} \mid \text{mean}\{T, T\} \mid \text{mean}\{T, T, T\} \\
C &\leftarrow 0 \mid 1 \mid \cdots \mid 9 \\
V &\leftarrow n_1 \mid n_2 \mid n_3 \mid n_4
\end{aligned}
$$

This grammar is able to represent addition, multiplication, various statistical functions taking two parameters (e.g., max), as well as a three-parameter mean operator, the numbers between 0 and 9 and the heuristic values of the current state's neighbours (i.e., n_1, n_2, n_3, n_4). These heuristic values are sorted in ascending order, meaning $n_i \leq n_{i+1}$ with n_1 being the lowest heuristic score of the neighbourhood.

4.1.3 Our Space

The grammar induces a set of functions $\tau(S, E, s, h)$ that receive as input a set of nodes S and edges E, the current state s_t of the agent and the current heuristic h_t.[3] As highlighted in Algorithm 3, we replace the learning rule on line 9 with the function: $\mathrm{h}_{t+1}(s_t) \leftarrow \tau(S, E, s_t, h_t)(4.0)$

We also include backtracking and depression avoidance as binary flags. They cannot be represented within the grammar as they are not a part of the learning rule. The new space of algorithms is $A = R \times \{\text{false}, \text{true}\} \times \{\text{false}, \text{true}\}$ where $R = \{\tau : \tau \text{ representable using the CFG}\}$. As shown in Chapter 5, this space includes a subset of the space of parameterised LRTA* algorithms (Bulitko, 2016a), as well as other algorithms previously unrepresentable.

[2] The tokens of this language are implicitly defined as any existing within a production rule. For example, T, n_2, and 3 are all tokens in this language.

[3] The search graph is required by the function in order to compute the neighbours of the current state and the heuristic is required when evaluating the tokens $\{n_1, n_2, n_3, n_4\}$.

Algorithm 3: Parameterised LRTA* with context-free grammar

Input: problem $p = (S, E, c, s_0, s_g, h)$, parameters w, b, *lop*, *bt*, *da*
Output: path $\psi = (s_0, s_1, \ldots, s_g)$

$t \leftarrow 0$
$\psi \leftarrow (s_0)$
while $s_t \neq s_g$ **do**
  if *da* **then**
    $\mu \leftarrow \min_{s \in N(s_t)} |h_0(s) - h_t(s)|$
    $\widetilde{N}(s_t) \leftarrow \{s \in N(s_t) : |h_0(s) - h_t(s)| = \mu\}$
  else
    $\widetilde{N}(s_t) \leftarrow N(s_t)$
  $h_{t+1}(s_t) \leftarrow \tau(S, E, s_t, h_t)$
  if *bt* & $h_{t+1}(s_t) > h_t(s_t)$ & $t > 0$ **then**
    $s_{t+1} \leftarrow s_{t-1}$
  else
    $s_{t+1} \leftarrow \arg\min_{s \in N(s_t)} c(s_t, s) + h_t(s)$
  $\psi \leftarrow \text{append}(\psi, s_{t+1})$
  $t \leftarrow t + 1$

4.2 Suboptimality

Our goal is to find an $a \in A$ which minimises $\alpha(a, P)$ given a set of problems $P = \{p_1, p_2, \ldots, p_n\}$. However, the effort to compute $\alpha(a, P)$ depends in part on $|P|$. Thus, we approximate $\alpha(a, P)$ as $\alpha(a, \hat{P})$ where $|\hat{P}| \ll |P|$. The suboptimality over $\hat{P}$, the *surrogate set*, is the *surrogate suboptimality* which serves as an estimate of the suboptimality of P. A large surrogate set more closely estimates the suboptimality of P but also makes $\alpha(a, \hat{P})$ more expensive to calculate. A small surrogate set can give a less accurate estimate but takes less time to compute. The surrogate suboptimality offers a trade-off between accuracy and speed. We examine these trade-offs empirically in Chapter 6.

Algorithm 4: Program Synthesis

Input: training set P, surrogate size s, budget m, cutoff $\alpha_{\max}$
Output: synthesised algorithm a_{best}

```
1  f_best ← ∞
2  P̂ ~^s P
3  i ← 0
4  while i ≤ m do
5      a ~ A
6      f_a ← α(a, P̂)
7      if f_a ≤ f_best then
8          a_best ← a
9          f_best ← f_a
10     i ← i + E(a, P̂)
```

4.3 Program Synthesis

With both the space of algorithms and their performance measure defined, we are able to synthesise an algorithm by searching for it in the space. The program synthesis runs for a finite amount of time and returns the best algorithm found.

For synthesis, we use simple stochastic search (Algorithm 4). The process is stochastic due to the use of random sampling, denoted by $x \sim X$ which draws a single random sample from a space, or $x \overset{n}{\sim} X$ which draws n random samples with replacement. Here we overload the variable x to mean a single sample or a set of samples. Algorithm 4 accepts as input a set of real-time heuristic search problems P, a surrogate set size s, a synthesis budget m and a suboptimality cutoff $\alpha_{\max}$. On line 1, we initialise the best suboptimality score. On line 2, we sample s problems to create a surrogate set $\hat{P}$. The search then runs until the synthesis budget has been exceeded (line 4). On line 5, we sample a single algorithm from the space of all possible algorithms. Line 6 then calculates the suboptimality of the sampled algorithm over the surrogate set of problems. On lines 7 through 9, if the surrogate suboptimality of the current

algorithm was the best seen so far, we update both the best seen algorithm and the best suboptimality score. Finally, i is incremented by the number of states expanded over the surrogate set. The synthesis returns the algorithm a_{best}.

Chapter 5

Theoretical Analysis

We will show that our space A defined by the context-free grammar (Chapter 4) covers an important subset of the space of parameterised LRTA* (Algorithm 2) algorithms. We first show an example of two algorithms: one which PLRTA* can represent and the CFG cannot, and the other an algorithm which the CFG can represent but PLRTA* cannot. We then prove that the context-free grammar covers a portion of the PLRTA* space when using the min operator. Finally, we provide an example of a PLRTA* algorithm that the CFG space can represent but which is not included in the proof.

Our space of context-free grammar algorithms is not a superset of the space of parameterised LRTA* algorithms (Figure 5.1). For example, the

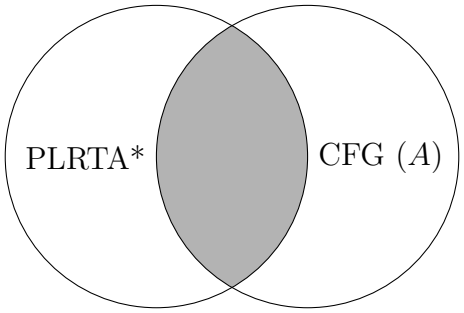

Figure 5.1: The PLRTA* and CFG space of algorithms intersect, but neither are strict supersets.

grammar is not able to produce floating point numbers. As such, the following parameterised LRTA* algorithm cannot be represented:

$$3.1 \cdot \min_{s \in \widetilde{N}_4 f(s_t)} (1 + h_t(s)) \tag{5.1}$$

The grammar also does not wholly represent the neighbourhood size restriction b for all learning operators. In order to allow for the arbitrary evaluation of $\{n_1, n_2, n_3, n_4\}$, we treat impassable terrain as portions of the neighbourhood with a maximum heuristic value. While this leaves the min operator equivalent, mean and max are not equivalent to parameterised LRTA* for non-constant neighbourhood sizes. Finally, due to our restriction to representing only integer variables, we use the definitions $\text{mean}(a, b) = \left\lfloor \frac{a+b}{2} \right\rfloor$ and $\text{mean}(a, b, c) = \left\lfloor \frac{a+b+c}{3} \right\rfloor$ and exclude median.

Conversely, we are able to represent a number of algorithms using the grammar that are unrepresentable in parameterised LRTA*. For example,

$$n_1 + 2 \cdot n_2 \tag{5.2}$$

cannot be represented in PLRTA*. Since parameterised LRTA* rules must contain $c(s_t, s)$ and $n_1 + 2 \cdot n_2$ does not contain $c(s_t, s)$ (i.e., a constant positive integer), A contains learning rules not found in parameterised LRTA*.

We now prove that our grammar is able generate any learning rule representable in PLRTA*, subject to the following restrictions: $w \in \mathbb{N} \cup \{0\}$ and $lop = \min$.

Theorem 1. The learning rule $w \cdot \min_{s \in \widetilde{N}_b^f(s_t)} (c(s_t, s) + h_t(s))$, restricted to $w \in \mathbb{N} \cup \{0\}$, can be represented by our context-free grammar. We assume the cost function $c(s_t, s)$ to be a constant integer.

We prove this by breaking the learning rule into its constituent parts and show that the grammar is able to represent each part.

Lemma 1. The multiplication of min by w can be encoded in the grammar.

Proof. This is achieved using the production rule $O \leftarrow C \times T$, as long as T is able to encode $\min_{s \in \widetilde{N}_b^f(s_t)}(c(s_t, s) + h_t(s))$. □

Lemma 2. The grammar can represent the learning operator min applied over the partial neighbourhood $\widetilde{N}_b^f$, which contains between one and four states.

Proof. Without loss of generality, assume the four possible states' heuristic values are a, b, c and d. For a neighbourhood with a single heuristic value a, we evaluate the learning operator over the neighbourhood using $\min(a, a)$. The case of a neighbourhood of size two is proven similarly $\min(a, b)$. The case of a neighbourhood with three heuristic values for min is represented using $\min(\min(a, b), c)$. Finally, a neighbourhood with four heuristic values can be combined using $\min(\min(a, b), \min(c, d))$. Since we restrict our proof to the min operator, this holds for $b \in \mathbb{N}$ as we only ever calculate the minimum heuristic value in the neighbourhood.[1] □

Lemma 3. The grammar can represent $\widetilde{N}_b^f(c(s_t, s) + h_t(s))$ when $|N| \leq 4$.

Proof. Using the rule $V \leftarrow n_1 \mid n_2 \mid n_3 \mid n_4$ we are able to select the neighbour with the i^{th} lowest heuristic score. We produce the f score using the rule $C + n_i$ where $C = c(s_t, s)$. Since $c(s_t, s)$ is a constant positive integer, the grammar can represent the cost using the integers 0 through 9 and addition. For example, the learning rule of LRTA* can be represented using $1 + n_1$. □

Using these lemmas, we now show that all integer-based parameterised LRTA* learning rules using the min operator are representable by our context-free grammar.

[1] This does not hold for mean or max as calculations are affected by the introduction of walls with large heuristic values to the neighbourhood calculation.

Proof. Using Lemma 1 we construct the expression $w \times T$. We replace T with O using $T \leftarrow O$. With a learning operator *lop*, we use Lemma 2 to substitute O with some combination of the rule $O \leftarrow \min\{T, T\}$. The exact combination is determined by the number of heuristic values (one to four) in the neighbourhood, dictated by i in Lemma 3, where $i = |\widetilde{N}|$. Finally, we substitute the inner T values with the expression $C + n_i$ from Lemma 3. We are thus able to represent all integer-based parameterised LRTA* learning rules that use the min operator. □

Finally, we note that the above proof does not cover the entire intersection of both spaces. There exist a number of learning rules in parameterised LRTA* that do not use the min operator that can be represented by the context-free grammar. For example, the parameterised LRTA* algorithm

$$3 \cdot \text{mean}_{s \in \widetilde{N}_1 f(s_t)}(1 + h_t(s)) \tag{5.3}$$

can be represented in A as

$$3 \cdot \text{mean}(1 + n_1, 1 + n_1) \tag{5.4}$$

While A is not a superset of PLRTA*, there is a significant amount of overlap that allows us to compare the spaces using program synthesis.

Chapter 6

Empirical Evaluation

This chapter evaluates the solution approach we proposed in Chapter 4. In Section 6.1 we describe the video game benchmarks we will be using. In Section 6.2 we describe the setup of each experiment. In Section 6.3 we evaluate our synthesis over real-time heuristic search problems.

6.1 Video Game Pathfinding

We analyse our synthesis method using video game pathfinding problems which serve as our benchmark for measuring suboptimality and are a *de facto* standard when evaluating real-time heuristic search.

6.1.1 Maps

In line with previous work (Bulitko, 2016a, 2020; Bulitko et al., 2021), we use video game maps from the Moving AI dataset (Sturtevant, 2012). Specifically, we use five maps from *Dragon Age II*: `ht_mansion2`, `lt_gallowscourtyard`, `w_blightlands`, `w_sundermount` and `w_woundedcoast`. These maps are pictured in Figure 6.1 with white representing passable terrain and black representing impassable obstacles called walls. The maps were chosen to represent both indoor and outdoor environments. The map dimensions are 308×358, 642×514, 514×514,

Figure 6.1: Five Moving AI video game maps from *Dragon Age II* (BioWare, 2011). The maps are `ht_mansion2`, `lt_gallowscourtyard`, `w_blightlands`, `w_sundermount` and `w_woundedcoast`, respectively.

770×770 and 578×642 cells, respectively.

We treat each map as a four-connected grid of cells. Since a learning rule may reference a neighbouring state when it does not exist (for example, evaluating n_3 when surrounded by two wall cells) we assign each impassable cell a heuristic score of $2^{64} - 1$. To avoid inaccuracies with floating-point operations, we only allow for non-negative integer heuristics using unsigned integer arithmetic that is bounded at 0 and $2^{64} - 1$. The movement cost along any edge is 1. The initial heuristic h_0 for every problem is defined as the Manhattan distance from a state $s = (x_0, y_0)$ to the goal $s_g = (x_g, y_g)$:

$$h(s, s_g) = |x_0 - x_g| + |y_0 - y_g| \tag{6.1}$$

6.1.2 Problem Sets

For each map, we generate 50 thousand unique random search problems. These are generated by selecting random start and goal states within the largest connected component of each map. We denote these P_m, P_g, P_b, P_s and P_w with the subscript indicating the first letter of the map's name (after the underscore). The coverage of P_m can be seen in Figure 6.2 with red indicating start states and green indicating goal states.

We initially solve each problem p using the A* algorithm and record the optimal path cost $c^*(p)$ (Table 6.1). We also run the LRTA* algorithm with a cutoff of 1000 (chosen in line with previous work; Bulitko, 2016a), for each

Figure 6.2: The problem coverage of the map `ht_mansion2` with red as starts and green as goals.

Table 6.1: The optimal path cost $c^*(p)$ when solving each problem set with A*, averaged over the 50000 problems on each map. Standard deviations are listed.

Problem set	**Optimal path cost**
P_m	196.62 $\pm$ 109.73
P_g	162.41 $\pm$ 97.22
P_b	360.16 $\pm$ 270.12
P_s	310.32 $\pm$ 189.78
P_w	427.72 $\pm$ 228.62

problem set and record the results in Table 6.2. The suboptimality in this table is the path cost of LRTA* divided by the path cost of A*.

6.2 Synthesis Implementation

In this section we describe our usage of syntax trees to represent our context-free grammar, how we select the size of the problem set used to calculate surrogate suboptimality and the cross-validation used in our experiments.

Table 6.2: The suboptimality and state expansions when solving each problem set with LRTA*, averaged over the 50000 problems on each map. Standard deviations are listed.

Problem set	**Mean suboptimality**	**State expansions**
P_m	354.24 ± 346.75	$8.91 \times 10^4 \pm 9.75 \times 10^4$
P_g	177.49 ± 331.84	$4.68 \times 10^4 \pm 9.90 \times 10^4$
P_b	230.27 ± 338.18	$1.22 \times 10^5 \pm 1.94 \times 10^5$
P_s	424.94 ± 438.08	$1.77 \times 10^5 \pm 2.07 \times 10^5$
P_w	274.47 ± 355.74	$1.38 \times 10^5 \pm 2.03 \times 10^5$

6.2.1 Syntax Trees

We define the space of real-time heuristic search algorithms A as those whose learning rule is represented by the grammar in Chapter 4. We implement the learning rules produced by this grammar using syntax trees. Each tree is randomly sampled using a recursive process. To begin, a single root node is generated and randomly assigned the type *operator* (O), *constant* (C) or *variable* (V). If the type of node is O it is randomly assigned an operator from the set $\{+, \times, \min, \max, \text{mean}\}$. It is also assigned randomly generated children nodes. If the type of node is C it is randomly assigned an integer value between 0 and 9. If the type of node is V it is randomly assigned a variable from the set $\{n_1, n_2, n_3, n_4\}$. Nodes of type C or V have no children. The recursion is continued until all leaf nodes are constants or variables. Algorithms are composed of a randomly generated syntax tree and two randomly selected boolean variables for backtracking and depression avoidance.

In order to evaluate a tree as a learning rule, it is traversed in postorder. Constants return their integer value, variables return the value of the heuristic at the specified neighbour state and operators return their operation applied to their children. A learning rule for LRTA* represented as a tree is in Figure 6.3.

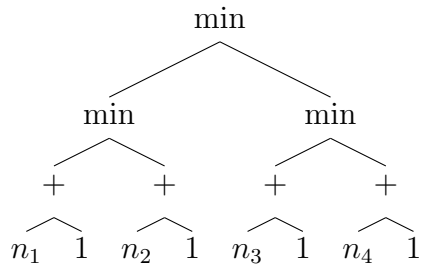

Figure 6.3: The LRTA* learning rule encoded as a syntax tree.

Table 6.3: The rank correlation (RC) and solution time for various surrogate set sizes sampled from P_m.

$\lvert\hat{\mathbf{P}}\rvert$	**RC**	$\hat{\mathbf{P}}$ **Time (s)**
25	0.62	0.34 $\pm$ 0.16
100	0.80	1.25 $\pm$ 0.48
250	0.76	3.44 $\pm$ 1.89
1000	0.90	13.11 $\pm$ 7.68

6.2.2 Surrogate Suboptimality

We determine the size of the surrogate set of problems empirically. To do so we draw 1000 random algorithms from the space A and calculate suboptimality over both P and $\hat{P}$, as well as the mean time required to calculate the surrogate suboptimality. Since our synthesis process depends only on the relative difference between suboptimalities, the surrogate suboptimality need not estimate the suboptimality of P. Rather, it is enough that the surrogate suboptimality maintains the relative ordering of the larger set's suboptimality values. In other words, it is enough that if $\alpha(a', \hat{P}) < \alpha(a'', \hat{P})$ then $\alpha(a', P) < \alpha(a'', P)$ for any algorithms a' and a''. We use Spearman's ρ (Spearman, 1904) to calculate the rank correlation. Higher values better represent the ordering of the suboptimalities of P. The results of this process are listed in Table 6.3. Since we are unable to achieve a RC of 1 in practice, we select a surrogate set size of 100 which has the RC of 0.8 and an average 1.25 second solution time.

6.2.3 Multiple Folds

We repeated the synthesis process over 5 folds as follows. We first split the problem set P into 5 disjoint partitions of the same size. We then ran 5 folds and, for each fold, grouped four partitions into P_{train} and one partition into P_{test}. These groupings were unique for each fold. The problem set P_{train} was passed into the synthesis algorithm. The resulting synthesised algorithms were then evaluated over P_{test}. Algorithm suboptimality calculated over P_{test} was then averaged over the 5 folds.

6.3 Analysis of the Results

In this section we list the hyperparameters used during synthesis, summarise the results of synthesis over both the parameterised LRTA* and context-free grammar space and analyse the portability of the synthesised algorithms.

6.3.1 Hyperparameters

Our empirical evaluation is controlled by a number of parameters listed in Table 6.4. The surrogate size $|\hat{P}|$ was set to 100 and the synthesis budget m was set to 10^{11} states expanded. These were based on preliminary experiments. The suboptimality cutoff $\alpha_{\max}$ was set to 1000, in keeping with existing work (Bulitko, 2016a). We compared two spaces of algorithms. The first was the parameterised LRTA* space of algorithms with $w \in [1, 2048]$, $b \in [0, 1]$, $lop \in \{\min, \max, \text{mean}\}$, $bt \in \{\text{false}, \text{true}\}$ and $da \in \{\text{false}, \text{true}\}$. The second was the space induced by our context-free grammar and the bt and da parameters.

Table 6.4: The hyperparameters used in our empirical evaluation.

Hyperparameters	**Value**
Problem set size ($\|P\|$)	5×10^4
Surrogate problem set size ($\|\hat{P}\|$)	100
Synthesis budget (m)	10^{11}
Cutoff ($\alpha_{\max}$)	10^3

Table 6.5: The synthesis results listing the mean suboptimality of the 5 folds as well as the mean synthesis time. Suboptimalities are bold when they outperform the opposing space's algorithm on the same problem set.

Space	**P**	**Average suboptimality**	**Synthesis time (h)**
P. LRTA*	P_m	**25.50** ± 9.60	3.32 ± 0.05
	P_g	33.19 ± 10.36	4.14 ± 0.07
	P_b	87.05 ± 47.73	3.37 ± 0.05
	P_s	100.39 ± 10.66	4.01 ± 0.03
	P_w	170.34 ± 61.16	3.50 ± 0.08
CFG	P_m	25.62 ± 9.01	6.03 ± 0.06
	P_g	**19.41** ± 12.60	6.94 ± 0.13
	P_b	**42.66** ± 19.38	6.16 ± 0.07
	P_s	**33.50** ± 13.70	6.76 ± 0.04
	P_w	**82.65** ± 51.75	6.17 ± 0.04

6.3.2 Results

Summary

Detailed results for the parameterised LRTA* (Table A.1) and context-free grammar (Table A.2) syntheses are listed in Appendix A. A summary of these results is in Table 6.5. Synthesis using the space induced by the context-free grammar outperformed synthesis using the parameterised LRTA* space on all problem sets except for P_m. The synthesis process was, however, about twice as long. This is because the randomly sampled parameterised LRTA* algorithms were faster to compute on average than those sampled from the context-free grammar space.

In Table 6.6 we present the context-free grammar algorithms for each

Table 6.6: The algorithms with lowest training suboptimality, manually simplified for clarity.

	Algorithm
a_m^{CFG}	$\text{mean}(n_1, (\text{mean}(0, n_2) \cdot 3))$
a_g^{CFG}	$5 + \min(n_4, \max(3, n_3))$
a_b^{CFG}	$\min(n_4, 6 + n_1)$
a_s^{CFG}	$\text{mean}(\text{mean}(\max(n_3, n_2 \cdot 4), 9 \cdot n_1, \max(n_4, 4)), n_2, n_1)$
a_w^{CFG}	$\min(n_1 \cdot \min(n_1, 4), \text{mean}(n_1, n_2) + 64) + 8$

problem set with the lowest single-fold training suboptimality, denoted by a_m^{CFG}, a_g^{CFG}, a_b^{CFG}, a_s^{CFG} and a_w^{CFG}. For clarity the algorithms in the table have been manually simplified. The original algorithms are presented in Appendix A. None of the algorithms can be represented using parameterised LRTA*. Furthermore, all of the algorithms take multiple n_i values into account, suggesting that there is a benefit to using more than just the minimum h score.

Synthesis over time

Test suboptimality over time of each individual fold is depicted in Figure 6.4. The x-axis denotes the states expanded by the synthesis process and the y-axis represents the test suboptimality of the best algorithm so far. This was calculated by maintaining a record of the best algorithm at each time step and evaluating them each on P_{test} once the synthesis completed. The evaluation of each algorithm on P_{test} was used only for plotting and did not influence the synthesis process. Each individual fold is represented by a dashed line and the mean of their interpolated values is represented by the thicker solid lines. The interpolation procedure is described in Appendix B. The context-free grammar (CFG) space is in blue while the parameterised LRTA* (PLRTA*) space is in red. The solid black line denotes the suboptimality of the LRTA* algorithm.

Algorithms synthesised in the CFG space achieved lower average

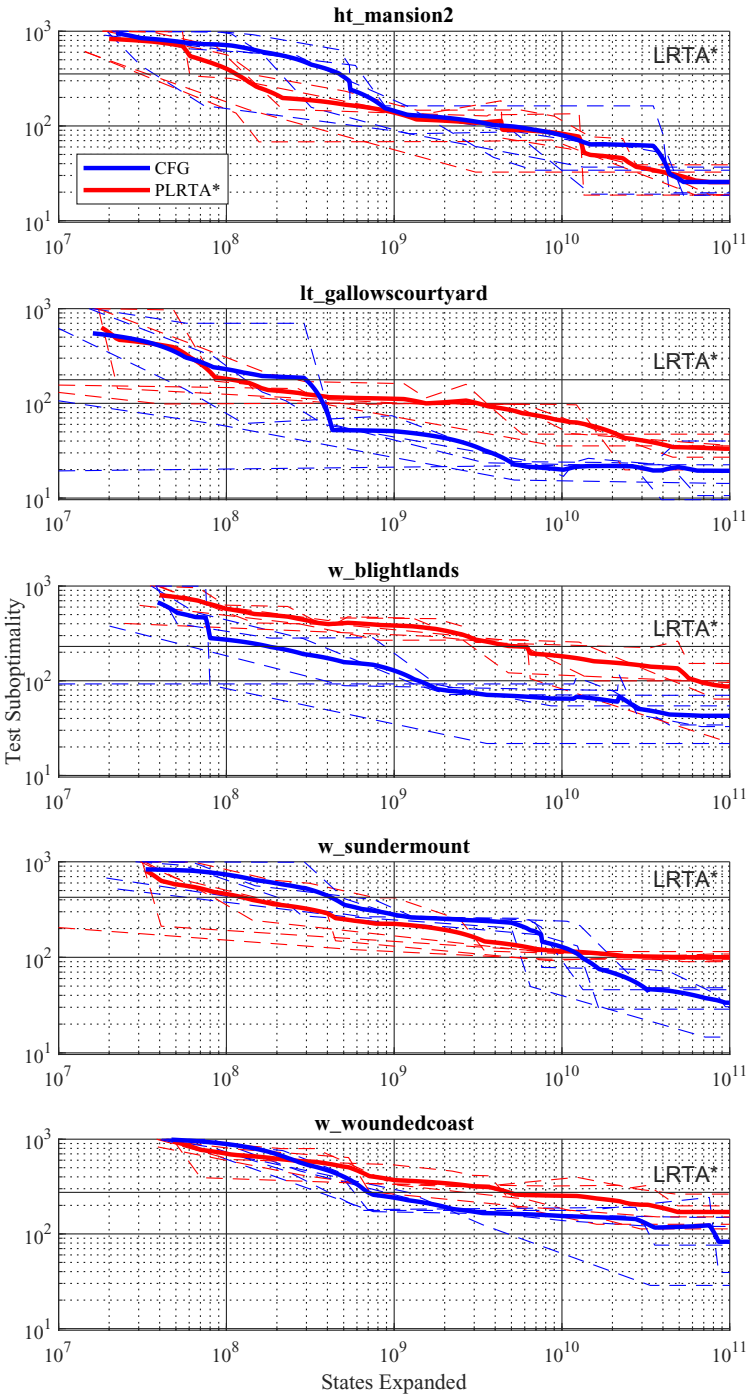

Figure 6.4: The suboptimality of each synthesised algorithm found as a function of the number of states expanded. Individual folds are shown with dashed lines and their interpolated averages are shown with thicker solid lines.

Table 6.7: The synthesis results when using an increased surrogate set size of $|\hat{P}| = 1000$. Suboptimalities are bold when they outperform the opposing space's algorithm on the same problem set.

Space	P	Average suboptimality	Synthesis time (h)
P. LRTA*	P_m	**64.44** ± 25.56	3.12 ± 0.07
	P_g	99.20 ± 6.29	3.74 ± 0.09
	P_b	270.21 ± 45.14	3.40 ± 0.04
	P_s	170.49 ± 66.346	4.28 ± 0.05
	P_w	257.06 ± 66.02	3.53 ± 0.06
CFG	P_m	88.00 ± 48.53	6.08 ± 0.07
	P_g	**22.95** ± 11.16	6.90 ± 0.15
	P_b	**89.98** ± 2.10	5.92 ± 0.08
	P_s	**104.58** ± 57.73	6.65 ± 0.08
	P_w	**180.99** ± 2.65	6.22 ± 0.04

suboptimalities than algorithms from the PLRTA* space on four problem sets. On P_b the average suboptimality of CFG algorithms always outperformed PLRTA* algorithms, although on P_g, P_s and P_w the PLRTA* algorithms initially outperformed the CFG algorithms. This supports our hypothesis that although the context-free grammar space of algorithms takes more effort to search, it represents a richer set of algorithms that eventually outperform parameterised LRTA*.

Increasing surrogate set size

We also ran the synthesis process with a larger surrogate set size. We chose $|\hat{P}| = 1000$, as it had the largest rank correlation in Table 6.3, while keeping all other hyperparameters the same. The results are shown in Table 6.7. Again, the CFG space outperformed the PLRTA* space on the same four problem sets. However, the resulting average suboptimalities were all higher when compared with those in Table 6.5. This is because using a larger surrogate size meant more states expanded per algorithm evaluation. As a result, fewer algorithms were evaluated over the course of the entire synthesis process.

Table 6.8: The suboptimality of algorithms evaluated over all problem sets. Bold indicates the lowest suboptimality for each problem set.

Algorithm	P_m	P_g	P_b	P_s	P_w
a_m^{CFG}	36.76	13.13	25.04	25.09	39.16
a_g^{CFG}	262.82	40.05	321.90	165.23	309.63
a_b^{CFG}	29.57	38.91	21.75	42.47	117.17
a_s^{CFG}	18.29	**9.28**	20.37	**14.63**	49.06
a_w^{CFG}	**12.47**	9.66	**16.99**	16.27	**28.59**

6.3.3 Portability

To investigate portability of the synthesised algorithms we evaluated them over the problem sets from maps on which they were not synthesised. We chose the algorithm from each map with the lowest single-fold training suboptimality (Table 6.6).

Each algorithm was evaluated over the set of 50 thousand problems for each map and their suboptimalities were recorded in Table 6.8. Aside from a_g^{CFG}, every algorithm outperformed LRTA* on all problem sets. a_s^{CFG} had the lowest suboptimality on two problem sets while a_w^{CFG} had the lowest suboptimality on the remaining three. These two algorithms are depicted in Figures 6.5 and 6.6 as syntax trees (neither contained backtracking or depression avoidance). The suboptimality of most algorithms (excluding the lowest performing a_g^{CFG}) remained relatively constant for all maps, suggesting that these algorithms are portable to problem sets not seen during synthesis.

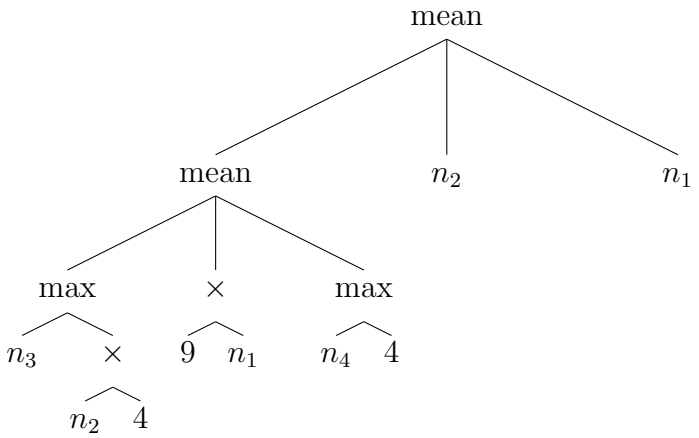

Figure 6.5: The algorithm a_s^{CFG}, which did not use backtracking or depression avoidance.

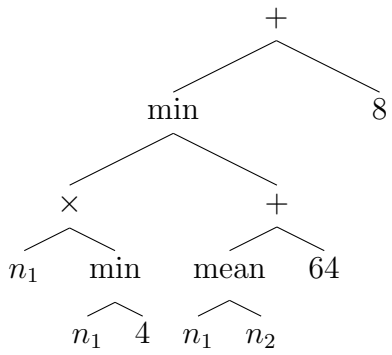

Figure 6.6: The algorithm a_w^{CFG}, which did not use backtracking or depression avoidance.

Chapter 7

Future Work

Our synthesis method randomly sampled algorithms and returned the one with the lowest training suboptimality. Further improvements could be made by replacing the random sampling with a more advanced method such as simulated annealing (Kirkpatrick et al., 1983), genetic algorithms (Barricelli, 1957; Eiben & Smith, 2015) or tabu search (Glover, 1986).

Our algorithms were determined mostly by the grammar used. Extensions to the grammar might add division or median operators. Other variables, such as the number of times a state has been visited, can also be considered.

If the synthesis time can be reduced by several orders of magnitude (i.e., from hours to seconds) then it may be feasible to automatically create RTHS algorithms for dynamic game maps. A further reduction of synthesis time might allow an agent to synthesise its own algorithm on the fly.

Future work will evaluate the synthesis method on other maps from the Moving AI dataset. The synthesis method can also be applied to any problem that is solvable using real-time heuristic search. For example, it would be interesting to note the algorithms generated for the sliding tile puzzle (Korf, 1990).

Finally, the synthesis could also be applied to non-real-time search, such as A*. Priority functions of A* (Chen & Sturtevant, 2019) could be represented

as a context-free grammar with the addition of a branching expression. Since A* finds optimal solutions, suboptimality would no longer be as useful of a metric, but recent work on generating non-real-time heuristics implemented loss as a function of state expansions (Bulitko et al., 2021), which could serve as a function to minimise.

Chapter 8

Conclusion

In this thesis, we proposed and evaluated a program synthesis over a new space of real-time heuristic search algorithms. Building upon existing work, we defined our space of learning rules using a context-free grammar. The algorithms generated using program synthesis outperformed rules synthesised within a subset of a published representation (Bulitko, 2016a) on several *Dragon Age II* maps. The synthesised algorithms also shown portability, achieving low suboptimalities on novel maps. We believe the work presented here is a stepping stone on the way to automatically synthesising more powerful heuristic search algorithms.

References

Barricelli, N. A. (1957). Symbiogenetic evolution processes realized by artificial methods. *Methodos*, *9*(35–36), 143–182.

BioWare. (2011). Dragon Age II.

Bodyul, V. (2010). Path Finding Visualizer. https://bodyulcg.com/tools/path-finding-visualizer/

Boyd, S. P., & Vandenberghe, L. (2014). *Convex Optimization*. Cambridge University Press.

Bulitko, V. (2004). Learning for Adaptive Real-time Search. *Computing Research Repository*, *cs.AI/0407016*.

Bulitko, V. (2016a). Evolving Real-time Heuristic Search Algorithms. *Proceedings of the Conference on the Synthesis and Simulation of Living Systems (ALIFE)*, 108–115.

Bulitko, V. (2016b). Searching for Real-time Heuristic Search Algorithms. *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 121–122.

Bulitko, V. (2020). Evolving Initial Heuristic Functions for Agent-Centered Heuristic Search. *Proceedings of the IEEE Conference on Games (CoG)*, 534–541.

Bulitko, V., & Doucet, K. (2018). Anxious Learning in Real-Time Heuristic Search. *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 1–4.

Bulitko, V., Hernandez, S. P., & Lelis, L. H. S. (2021). Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding. *Proceedings of the IEEE Conference on Games (CoG)*, in press.

Bulitko, V., & Sampley, A. (2016). Weighted Lateral Learning in Real-time Heuristic Search. *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 10–18.

Chen, J., & Sturtevant, N. R. (2019). Conditions for Avoiding Node Re-expansions in Bounded Suboptimal Search. *Proceedings of the Joint Conference on Artificial Intelligence (IJCAI)*, 1220–1226.

Eiben, A. E., & Smith, J. E. (2015). *Introduction to evolutionary computing*. Springer.

Glover, F. W. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, *13*(5), 533–549.

Gulwani, S., Polozov, O., & Singh, R. (2017). Program Synthesis. *Foundations and Trends in Programming Languages*, *4*(1-2), 1–119.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Hendrikx, M., Meijer, S. A., Velden, J. V. D., & Iosup, A. (2013). Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*, *9*(1), 1–22.

Hernandez, S. P., & Bulitko, V. (2021). Speeding Up Heuristic Function Synthesis via Extending the Formula Grammar. *Proceedings of the Symposium on Combinatorial Search (SoCS)*, in press.

Hernández, C., & Baier, J. A. (2012). Avoiding and Escaping Depressions in Real-Time Heuristic Search. *Journal of Artificial Intelligence Research*, *43*, 523–570.

Hernández, C., Botea, A., Baier, J. A., & Bulitko, V. (2017). Online Bridged Pruning for Real-Time Search with Arbitrary Lookaheads. *Proceedings of the Joint Conference on Artificial Intelligence (IJCAI)*, 510–516.

Hernández, C., & Meseguer, P. (2005). LRTA*(k). *Proceedings of the Joint Conference on Artificial Intelligence (IJCAI)*, 1238–1243.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Koenig, S. (2001). Agent-Centered Search. *AI Magazine*, *22*(4), 109–132.

Koenig, S., & Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, *18*(3), 313–341.

Korf, R. E. (1990). Real-Time Heuristic Search. *Artificial Intelligence*, *42*(2-3), 189–211.

Manna, Z., & Waldinger, R. J. (1971). Towards automatic program synthesis. In *Proceedings of the Symposium on Semantics of Algorithmic Languages* (pp. 270–310).

Muñoz, F., Fadic, M., Hernández, C., & Baier, J. A. (2018). A Neural Network for Decision Making in Real-Time Heuristic Search. *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 173–177.

Rivera, N., Baier, J. A., & Hernández, C. (2015). Incorporating weights into real-time heuristic search. *Artificial Intelligence*, *225*, 1–23.

Sharon, G., Sturtevant, N. R., & Felner, A. (2013). Online Detection of Dead States in Real-Time Agent-Centered Search. *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 167–174.

Spearman, C. (1904). The Proof and Measurement of Association Between Two Things. *American Journal of Psychology*, *15*, 88–103.

Sturtevant, N. R. (2012). Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144–148.

Sturtevant, N. R., & Bulitko, V. (2014). Reaching the Goal in Real-Time Heuristic Search: Scrubbing Behavior is Unavoidable. *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 166–174.

Sturtevant, N. R., & Bulitko, V. (2016). Scrubbing During Learning In Real-time Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, *57*, 307–343.

Sturtevant, N. R., & Geisberger, R. (2010). A Comparison of High-Level Approaches for Speeding Up Pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 76–82.

Appendix A

Detailed Results

Detailed results of the synthesis processes in Chapter 6 follow. Each entry contains the problem set, the fold number, the resulting synthesised algorithm, the test suboptimality of that algorithm and the total synthesis time of the synthesis process. The results for parameterised LRTA* are in Table A.1 and the results for our context-free grammar in Table A.2. Bold numbers indicate the lowest test suboptimality of the 5 folds for a given problem set.

Table A.1: The results of program synthesis over the parameterised LRTA* space.

P	Fold	Synthesised Algorithm	Test Suboptimality	Synthesis Time (h)
P_m	1	$17 \cdot \text{mean}_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	32.57	3.30
	2	$5 \cdot \min_{s \in \widetilde{N}_2^f}(1 + h_t(s))$	38.89	3.35
	3	$5 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	18.70	3.27
	4	$6 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	**18.62**	3.30
	5	$4 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s))$	18.74	3.40
P_g	1	$9 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	36.25	4.20
	2	$8 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	27.01	4.06
	3	$4 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	**19.86**	4.13
	4	$9 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	35.56	4.23
	5	$10 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	47.28	4.09
P_b	1	$2 \cdot \text{mean}_{s \in \widetilde{N}_1^f}(1 + h_t(s))$	64.08	3.45
	2	$12 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	99.77	3.32
	3	$24 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s))$	152.30	3.33
	4	$9 \cdot \text{mean}_{s \in \widetilde{N}_3^f}(1 + h_t(s))$	95.96	3.36
	5	$4 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	**23.12**	3.39
P_s	1	$13 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s)) + \text{da}$	95.58	3.98
	2	$9 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s)) + \text{da}$	93.16	3.99
	3	$8 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s))$	108.25	4.05
	4	$4 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s))$	**90.07**	4.00
	5	$56 \cdot \text{mean}_{s \in \widetilde{N}_3^f}(1 + h_t(s)) + \text{da}$	114.91	4.04
P_w	1	$8 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	**113.02**	3.52
	2	$32 \cdot \min_{s \in \widetilde{N}_2^f}(1 + h_t(s))$	263.21	3.43
	3	$10 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	126.32	3.57
	4	$14 \cdot \min_{s \in \widetilde{N}_4^f}(1 + h_t(s))$	151.37	3.57
	5	$21 \cdot \min_{s \in \widetilde{N}_3^f}(1 + h_t(s))$	197.78	3.41

Table A.2: The results of program synthesis over the context-free grammar space.

P	Fold	Synthesised Algorithm	Test Suboptimality	Synthesis Time (h)
P_m	1	$\text{mean}(n_2, \text{mean}(n_3, \max(n_1, (n_1 \cdot n_1)), n_1), \min(n_1, n_1))$	19.69	6.09
	2	$\text{mean}((((\min(n_2, n_1) + \max(\min(\text{mean}(n_1, (9 \cdot n_4), n_3), n_2), n_1)) + n_1) + 8), 4, 4)$	19.04	5.93
	3	$\text{mean}(n_1, (\text{mean}(0, n_2) \cdot 3))$	36.76	6.07
	4	$\min(((\min(1, 3) \cdot \text{mean}(3, 8)) \cdot n_2), \min(n_3, (n_2 + \max((\max(3, n_1) + 7), 8))))$	**18.54**	6.04
	5	$(4 + \min(n_3, n_1))$	34.09	6.04
P_g	1	$(3 + n_2)$	22.47	6.95
	2	$(5 + \min(n_4, \max(3, n_3)))$	40.05	7.09
	3	$\text{mean}(\text{mean}(\text{mean}(\max(2, (\max(n_3, n_1) \cdot n_4)), n_1, 3), n_1), \text{mean}(2, \min(1, \min((n_1 \cdot \text{mean}(\max(n_3, (n_3 + n_1)), 6)), (n_1 + n_2))), (3 \cdot 6)), n_3)$	**9.62**	6.92
	4	$\min(n_3, (2 \cdot n_2))$	14.28	6.74
	5	$\min(\text{mean}(n_1, n_3, (n_2 + \min(n_3, (4 \cdot \max(1, n_2))))), (n_4 \cdot n_4))$	10.62	6.97
P_b	1	$\max(\max(4, \text{mean}(2, n_4)), n_1)$	70.23	6.14
	2	$(2 + \min(n_1, (n_1 + (n_3 + 8))))$	54.34	6.12
	3	$(\min(2, 8) + \text{mean}(n_1, \text{mean}(\max(7, n_1), \max(1, (n_2 \cdot 5)))))$	34.16	6.22
	4	$\min(n_4, (6 + n_1))$	**21.75**	6.08
	5	$\min(\text{mean}((\max(5, 4) \cdot n_2), n_1, n_2), n_4)$	32.84	6.23
P_s	1	$\text{mean}(\text{mean}(\max(n_3, (n_2 \cdot 4)), (9 \cdot n_1), \max(n_4, 4)), n_2, n_1)$	**14.63**	6.78
	2	$(\min((n_3 \cdot n_2), (\min(\max(\text{mean}(n_1, n_4, 8), 4), n_3) + \min(n_3, 5))) \cdot 2)$	28.66	6.82
	3	$\max(n_2, \max(\min(n_1, 5), n_1))$	45.82	6.75
	4	$\min((n_1 + n_2), \text{mean}(\text{mean}(\max(\max(\max(0, \text{mean}(n_2, \text{mean}(n_3, 4), \max(8, 1))), 6), (\max(\text{mean}(\min(n_1, 1), \text{mean}(n_2, n_1), n_2), \text{mean}(n_3, 5, n_3)) \cdot 4)), 4, n_3), n_1))$	30.40	6.74
	5	$\max(n_2, n_1)$	48.00	6.71
P_w	1	$\text{mean}((8 + n_1), n_1, n_1)$	149.49	6.15
	2	$\min(n_3, (n_2 + \text{mean}(1, 3)))$	119.91	6.24
	3	$\text{mean}((n_1 + (n_2 + \text{mean}(\text{mean}(\max(n_3, 3), 2), 2, 2))), 0)$	76.11	6.14
	4	$(\min((n_1 \cdot \min(n_1, 4)), (\text{mean}(n_1, n_2) + (8 \cdot 8))) + 8)$	**28.59**	6.16
	5	$\min(n_3, \max(\text{mean}(\text{mean}(((n_3 + n_1) \cdot \max(8, ((2 + 9) \cdot (7 \cdot 3)))), n_3), n_1, 7), 2))$	39.16	6.17

Appendix B

Interpolation

We describe the interpolation of the results discussed in Chapter 6. The interpolation occurred over 5 folds. Since each fold recorded the number of states expanded and test suboptimality when the best algorithm was outperformed, each fold was not guaranteed to contain the same number of data points.

Consider two folds with the data points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_m, y_m)\}$ and $\{(x'_0, y'_0), (x'_1, y'_1), \ldots, (x'_n, y'_n)\}$ with $m \neq n$. Our goal is to calculate the pointwise average of both sets. However, since they are not of the same size and because x_i may not equal x'_i, we first generate intermediate points.

We generate q evenly spaced x values within each set. For the experiments in Chapter 6, we place an x value at every 10^5 state expansions such that both sets end up with an even number of x values $\{0, 10^5, 20^5, \ldots, \min(x_m, x_n)\}$. We denote these values $\{c_0, c_1, \ldots, c_q\}$.

The corresponding y values are constructed using a linear interpolation. Assuming, without loss of generality, we are working with the first fold and we wish to find a d_i for a given c_i. Assuming $x_{j-1} \leq c_i \leq x_j$ for some j, then d_i is the linear interpolation of y_{j-1} and y_j. The first fold is left with the points $\{(c_0, d_0), (c_1, d_1), \ldots, (c_q, d_q)\}$. We repeat for each fold. Since each fold now contains an equal amount of datapoints, we calculate the pointwise average.

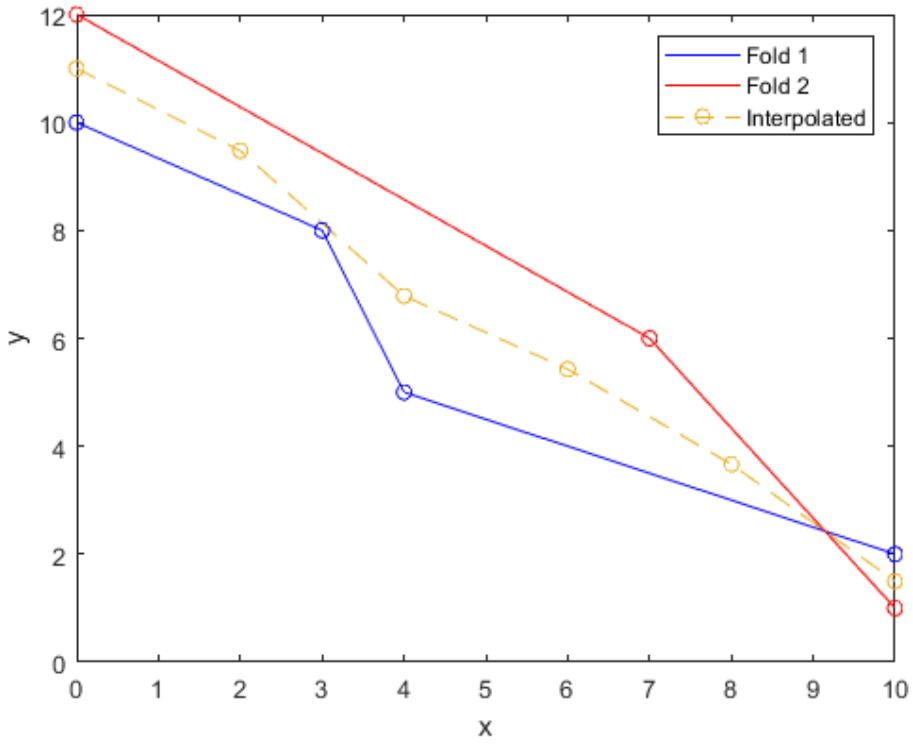

Figure B.1: The data points of both example folds and their interpolated average.

For example, assume we have two folds containing the data points $\{(0, 10), (3, 8), (4, 5), (10, 2)\}$ and $\{(0, 12), (7, 6), (10, 1)\}$. In Figure B.1, the first set is depicted in blue and the second in red. We interpolate every second x value which results in the final interpolated points $\{(0, 11), (2, 9.48), (4, 6.79), (6, 5.42), (8, 3.67), (10, 1.50)\}$, shown in orange.