Proceedings of the Fifteenth International Symposium on Combinatorial Search (SoCS 2022)

Portability and Explainability of Synthesized Formula-based Heuristics

Vadim Bulitko, Shuwei Wang, Justin Stevens, Levi H. S. Lelis

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada
{bulitko, shuwei4, jdsteven, santanad}@ualberta.ca

Abstract

Heuristic search is a key component of automated planning and pathfinding. It is guided by a heuristic function which estimates remaining solution cost. Traditionally heuristic functions for pathfinding have been human-designed or pre-computed for a specific search graph. The former tend to be compact, human-readable but generic. The latter offer better guidance but require per-graph pre-computation and have a substantial memory cost. We aim to retain compactness and readability of human-designed heuristics and increase their performance. We adopt the recently published approach of representing heuristic functions as algebraic formulae and automatically synthesizing them for video-game maps. Whereas published work merely randomly sampled the space of formula-based heuristic functions, we implement and evaluate a parameterized synthesis algorithm that unifies and generalizes the stochastic sampling, simulated annealing and a basic genetic algorithm. We tune the parameters for better synthesis performance and then, using maps from multiple video games, show that heuristics synthesized for maps from one game still outperform the baseline search (A* with weighted Manhattan distance) on maps from a different game. We analyze a frequently synthesized formula and explain how, despite having a higher error than the Manhattan distance, it takes advantage of the structure in video-game pathfinding problems and speeds up A*.

1 Introduction

Heuristic search is effectively used in automated planning. A classical heuristic search algorithm such as A* (Hart, Nilsson, and Raphael 1968) explores a given weighted graph to find a lowest-cost solution between two given vertices. In doing so A* explores the graph by expanding vertices and evaluating vertices in their immediate neighborhoods. The search effort is commonly measured in the number of states expanded. The exploration of the search graph is guided by a heuristic function (or a *heuristic*) which attempts to predict the remaining cost of a path from a given vertex to the goal vertex. Generally speaking, more accurate heuristics reduce the search effort by guiding the search algorithm in the promising direction. Thus efficiency of heuristic search substantially depends on the quality of the heuristic used, creating the problem of designing high-quality heuristics.

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Traditionally heuristics have been either manually designed or pre-computed for a specific graph (e.g., memory-based heuristics). The former tend to be simple and generic which makes them applicable to a broad class of search graphs. However, their accuracy can be low due to their simple and generic nature which causes heuristic search algorithms to expend extra effort while searching. On the other hand, heuristics pre-computed for specific search graphs yield good search performance by capturing specifics of a graph (Björnsson and Halldórsson 2006; Sturtevant et al. 2009; Culberson and Schaeffer 1998; Korf and Felner 2002; Felner et al. 2007; Korf 1997; Edelkamp 2001). The downside is their lack of portability and a large memory cost.

Recent work (Bulitko, Hernandez, and Lelis 2021) attempted to combine the simplicity and portability of manually designed heuristics with the higher search performance of per-graph pre-computed heuristics. It did so by synthesizing formula-based heuristics which can be complex enough to potentially capture some specifics of a search graph but still simple enough to have a negligible memory cost and effectively apply to different graphs. The simplicity of synthesized heuristics was achieved *implicitly* through (i) defining a space of heuristics as algebraic formulae generated by a compact context-free grammar and (ii) sampling the space randomly. Each sampled formula was generated by mutating a random single-terminal formula up to a fixed number of times. The randomly chosen mutation operations could both grow the formula and shrink it which effectively limited the overall size and complexity of the resulting formulae.

This paper builds on the published work (Bulitko, Hernandez, and Lelis 2021) and makes the following **contributions**. First, we implemented a parameterized algorithm which unifies and generalizes the stochastic sampling, simulated annealing and a genetic algorithm. The latter two allow the synthesis process to *accumulate* mutations over time, potentially producing larger and more complex heuristic formula which can capture more specifics of a search graph. The generalized synthesis algorithm is enhanced with progressive candidate evaluation, called *triage* by Bulitko, Hernandez, and Lelis (2021).

Second, we illustrate how a commonly synthesized heuristic makes A* expand substantially fewer states than the Manhattan distance despite having a higher error relative to the perfect heuristic. We also demonstrate how the

formula-based representation of a synthesized heuristic can enable a straightfoward way to establish a solution suboptimality bound. The third contribution of this paper is an empirical study of portability of synthesized heuristics across different search graphs. Unlike the published work (Bulitko, Hernandez, and Lelis 2021), we used two different classes of video-game maps as our search graphs and considered how heuristics synthesized for one class of maps can be helpful for another class. We showed that synthesized heuristics appear to capture and take advantage of the maps they were synthesized for. Yet, they are portable enough to outperform the baseline search even on maps from a different game.

Please contact us for our code (C++/MATLAB) and data.

2 Problem Formulation

We tackle the same synthesis problem as Bulitko, Hernandez, and Lelis (2021), reproduced below for the reader's convenience. We first define notation related to heuristic search on graphs and then frame synthesis of heuristic functions as an optimization problem.

A **graph search problem** p is defined by a tuple (G, s_0, s_g). Here $G = (S, E, c)$ is the search graph. A state $s_i \in S$ is connected to its neighbor s_j via an edge $(s_i, s_j) \in E$ with a *cost* $c(s_i, s_j) > 0$. The *neighborhood* of state s is $N(s) = \{s' \mid (s, s') \in E\}$. A state is *expanded* by a heuristic search algorithm when its $N(s)$ is computed and evaluated. The state s_0 is the *initial state* and s_g is the *goal state*. A solution to a graph search problem is a *path* $(s_0, s_1, \ldots, s_g)$ such that any pair $(s_i, s_{i+1}) \in E$. The cumulative cost of all edges on a solution path is the *solution cost*. The *solution suboptimality* α is the ratio of the solution cost to the lowest possible solution cost for problem p, denoted by $c^*(p)$. Solution suboptimality $\alpha = 1$ indicates an optimal solution while $\alpha = 2$ indicates a path twice as costly as optimal. Lower suboptimality values indicate lower-cost solutions and are thus preferred. Heuristic search algorithms employ a *heuristic function* h to guide their exploration of the search graph in an attempt to find a lower cost solution faster. The value of $h(s)$ is a cost estimate of the lowest-cost path from s to s_g.

The **synthesis problem** we tackle is a procedural generation of a heuristic function that maximizes search algorithm performance on a set of search problems. This can be viewed as an optimization problem of minimizing a loss function $h_{\min} = \operatorname*{argmin}_{h \in H} \ell(a, h, P)$ where a is a search algorithm, H is a space of heuristic functions, P is a set of search problems and ℓ is a *loss function*. For instance, the set P can consist of all pathfinding problems on a given video-game map. The loss ℓ is the ratio of the average number of states expanded using h to the average number of states expanded by a baseline. To illustrate, a (non-regularized) loss of 0.5 means that the search algorithm a with the heuristic h expanded half the nodes expanded by a with the baseline heuristic while finding solutions of the same quality (so a speed-up of 2 times). Here we used the same interpolation and averaging process as Bulitko, Hernandez, and Lelis (2021). Note that during the synthesis process we regularize the loss to bias the synthesis towards more compact heuristic formula. Post-synthesis we compute the test loss without regularization.

Additionally we will study how robust a heuristic synthesized for one set of problems is with respect to a different set of problems. Formally, we define *degradation* of heuristic h performance as $\mathcal{D}(a, P, h_P, P', h_{P'}) = \ell(h_P, a, P') - \ell(h_{P'}, a, P')$ where the heuristic h_P is synthesized for the problem set P but its loss $\ell(h_P, a, P')$ is computed on a different problem set P' and is compared to the loss $\ell(h_{P'}, a, P')$ of a heuristic $h_{P'}$ synthesized *specifically* for P'. Low degradation means high *portability* of the heuristic.

We **prefer** a solution to the synthesis problem formulated here which is automated, can reasonably run on a single commodity computer but can also take advantage of a cluster, produces short human-readable heuristics with low loss and low degradation across video-game maps.

3 Related Work

The search space of heuristic functions lacks a factored representation such as the space of parameters of a generalized LRTA* algorithm (Bulitko 2016). Consequently, the tabulated parameter sweep used by Bulitko (2016) would not apply. A popular program synthesis approach, bottom-up search, uses small programs as building blocks for larger programs (Albarghouthi, Gulwani, and Kincaid 2013; Udupa et al. 2013). Uninformed bottom-up search generates all programs of a certain size before generating programs of a larger size. In practice the grammar defining the program space can have a large branching factor (e.g., our grammar has 97 terminal symbols) which renders synthesizing even a moderately sized program in the bottom-up fashion intractable both time- and memory-wise. Informed bottom-up search can require days of training on GPU (Shi et al. 2021) while we desire to synthesize high-performance heuristics in at most a few hours. Additionally Shi, Bieber, and Singh (2020), Odena et al. (2020) and Shi et al. (2021) considered program synthesis with the task specification given by input/output pairs which we do not have.

Recent work on heuristic synthesis has used a simple random sampling (Bulitko, Hernandez, and Lelis 2021; Hernandez and Bulitko 2021) and a basic genetic algorithm (Bulitko 2020). The former is simple and fast but does not accumulate mutations over time and is thus unable to incrementally build up complexity of the synthesized heuristics. The latter can accumulate mutations but can produce very long formulae which are difficult to interpret and analyze. Additionally the genetic algorithm has more hyperparameters requiring more domain-specific tuning.

4 Our Approach

We address the problems with related work as follows. To avoid the high computational complexity of the systematic bottom-up search we explore the space of heuristic formulae stochastically. To enable more complex synthesized formulae we allow accumulation of mutations. To prevent the formulae from becoming too complex we regularize their performance loss with the formulae size. Finally, to include

previous algorithms in the study we propose a simple synthesis algorithm that unifies and generalizes the previously used stochastic sampling, simulated annealing with history and a basic genetic algorithm (Adler 1993). It is controlled by hyperparameters which we tune algorithmically for the domain at hand which allows us to explore effects of the control parameters (e.g., the temperature in the simulated annealing) beyond the original algorithms. This can be viewed as a metaheuristic optimization (Yang 2011).

The proposed synthesis algorithm thus combines the following six techniques: (i) *stochastic sampling* of the formula space (Bulitko, Hernandez, and Lelis 2021; Hernandez and Bulitko 2021), (ii) *accumulating mutations* (Kirkpatrick, Gelatt, and Vecchi 1983; Bulitko 2020), (iii) *probabilistic acceptance* of candidates (Kirkpatrick, Gelatt, and Vecchi 1983; Alur et al. 2013), (iv) maintaining a *population of promising formulae* (Bulitko 2020), (v) *progressive evaluation* of candidates (Bulitko, Hernandez, and Lelis 2021) and (vi) using previously synthesized formulae as *building blocks* (Bulitko, Hernandez, and Lelis 2021; Albarghouthi, Gulwani, and Kincaid 2013; Udupa et al. 2013).

4.1 Multiple Synthesis Trials

As Bulitko, Hernandez, and Lelis (2021), we parallelize the synthesis at multiple scales. Each candidate heuristic from the heuristic space H is evaluated with the search algorithm on multiple problems in parallel, taking advantage of multiple cores on a single computer. At the same time the synthesis process can be run multiple times or *trials* across several computers (e.g., cluster nodes) in parallel, followed by selecting the best synthesized heuristic among them.

We run T independent synthesis trials in parallel in line 1 of Algorithm 1. Each trial t calls the function `trial` which uses our synthesis algorithm (Section 4.2) to synthesize a heuristic h_t with its regularized loss l_t^λ measured on the training set P_{train3}. The output of the parallel process is then the synthesized heuristic h with the lowest loss (line 4).

We regularize the loss function with the size of the heuristic formula to control formula complexity and maintain human-readability as well as prevent overfitting to the set of training problems used during synthesis. Thus, instead of the loss $\ell(a, h, P)$ used in previously published work we use $\ell^\lambda(a, h, P) = \ell(a, h, P) + \lambda|h|$ where $|h|$ is the number of vertices in the syntax tree representing the heuristic h (Bulitko, Hernandez, and Lelis 2021) and λ is the regularizer constant. We omit λ when clear from the context.

Algorithm 1: Parallel multi-trial synthesis

input : training problem sets $P_{\text{train1}}, P_{\text{train2}}, P_{\text{train3}}$, heuristic space H, per-trial synthesis budget b, regularized loss function ℓ^λ, number of trials T, search algorithm a
output: synthesized heuristic h

1 **for** $t = 1, \ldots, T$ **in parallel do**
2 $\quad h_t \leftarrow \texttt{trial}(b, H, \ell^\lambda, P_{\text{train1}}, P_{\text{train2}}, a)$
3 $\quad l_t^\lambda \leftarrow \ell^\lambda(a, h_t, P_{\text{train3}})$
4 **return** $h = \operatorname{argmin}_t l_t^\lambda$

Algorithm 2: Single synthesis `trial`

input : training problem sets $P_{\text{train1}}, P_{\text{train2}}$, heuristic space H, synthesis budget b, loss function ℓ, regularizer λ, search algorithm a, population size n, number of champions c, number of survivors s, temperature τ, flag δ
output: the lowest synthesized heuristic seen $h_{\text{historic best}}$
assert : $c \le n, s \le c$

1 $h_{1,\ldots,n} \sim H$
2 $h^\diamond_{1,\ldots,c} \leftarrow h_{1,\ldots,c}$
3 $l^\diamond_{1,\ldots,c} \leftarrow (\infty, \ldots, \infty)$
4 $l_{\text{historic best}} \leftarrow \infty$
5 **repeat**
6 $\quad h'_{1,\ldots,c} \leftarrow \texttt{best-of}_{\ell^\lambda(a,h_i,P_{\text{train1}})}(h_{1,\ldots,n})$
7 $\quad h^\diamond_{1,\ldots,c} \leftarrow \texttt{sort}_{l^\diamond}(h^\diamond_{1,\ldots,c})$
8 $\quad$**for** $i = 1, \ldots, c$ **do**
9 $\quad\quad h^\diamond_{1,\ldots,c}, l^\diamond_{1,\ldots,c} \leftarrow \texttt{insert}(h^\diamond_{1,\ldots,c}, l^\diamond_{1,\ldots,c}, h'_i, \ell^\lambda(a, h'_i, P_{\text{train2}}), \tau)$
10 $\quad$**if** $\min_{i=1,\ldots,c} l^\diamond_i < l_{\text{historic best}}$ **then**
11 $\quad\quad l_{\text{historic best}} \leftarrow \min_{i=1,\ldots,c} l^\diamond_i$
12 $\quad\quad h_{\text{historic best}} \leftarrow h^\diamond_{\operatorname{argmin}_{i=1,\ldots,c} l^\diamond_i}$
13 $\quad$**if** δ **then**
14 $\quad\quad h_{1,\ldots,s} \leftarrow h^\diamond_{1,\ldots,s}$
15 $\quad\quad h_{s+1,\ldots,n} \leftarrow \texttt{offspring}(h^\diamond_{1,\ldots,c})$
16 $\quad$**else**
17 $\quad\quad h_{1,\ldots,n} \sim H$
18 $\quad$reduce τ
19 **until** *b is exhausted*
20 **return** $h_{\text{historic best}}$

4.2 A Single Synthesis Trial

Henceforth we use $x_{1,\ldots,k}$ as a shorthand for $(x_1, \ldots, x_k)$. We start with a population of n heuristics randomly drawn from the heuristic space H in line 1 of Algorithm 2. The algorithm maintains champions, c heuristics and their regularized loss values, as the collections $(h^\diamond_1, \ldots, h^\diamond_c)$ and $(l^\diamond_1, \ldots, l^\diamond_c)$ or $h^\diamond_{1,\ldots,c}$ and $l^\diamond_{1,\ldots,c}$ for short. Initially the champions are set to the first c members of the heuristic population (line 2) and their loss values are all set to ∞ (line 3).

Then, as long as the synthesis budget of b states expanded is not exhausted by running the search algorithm a, we run the following loop. First, in line 6 each heuristic h_i in the population $h_{1,\ldots,n}$ has its regularized loss $\ell^\lambda(a, h_i, P_{\text{train1}})$ computed on the small training set P_{train1}. Then the c heuristics with the lowest loss become candidates to be the new champions, denoted by $h'_{1,\ldots,c}$. This step implements filtering within triage of Bulitko, Hernandez, and Lelis (2021).

Second, in lines 8 – 9 we probabilistically merge the existing champions $h^\diamond_{1,\ldots,c}$ with the candidate champions $h'_{1,\ldots,c}$ based on their regularized loss computed on the larger training set of problems P_{train2}. We first sort the existing champions by their regularized loss $l^\diamond$ (line 7). Then we insert each candidate champion h'_i into the collection of existing champions in line 9. The insertion (Algorithm 3) takes the champions $h^\diamond_{1,\ldots,c}$ and a candidate champion h'_i with its regular-

ized loss $\ell^\lambda(a, h'_i, P_{\text{train2}})$ computed on the larger training set P_{train2}. It then probabilistically inserts it into the collection of champions by comparing it to j-th champion (line 2) (Zgierski 1993). If the candidate is accepted then it is inserted in front of the j-the champion (line 3) together with its loss (line 4). Note that the collection of champions retains its size of c elements. To do so we discard the worst (i.e., c-th) champion in C, $h^\diamond_c$, and its loss $l^\diamond_c$ upon each insertion. Thus the procedure resembles an insertion sort algorithm but with a probabilistic comparison and a bounded array size.

Algorithm 3: Probabilistic `insert`

input : champions $h^\diamond_{1,\dots,c}, l^\diamond_{1,\dots,c}$, candidate champion h and its regularized loss l, temperature τ
output: updated champions $h^\diamond_{1,\dots,c}, l^\diamond_{1,\dots,c}$

1 **for** $j = 1, \dots, c$ **do**
2   **if** `accept`$(l, l^\diamond_j, \tau)$ **then**
3     $h^\diamond_{1,\dots,c} \leftarrow (h^\diamond_1, \dots, h^\diamond_{j-1}, h, h^\diamond_j, h^\diamond_{j+1}, \dots, h^\diamond_{c-1})$
4     $l^\diamond_{1,\dots,c} \leftarrow (l^\diamond_1, \dots, l^\diamond_{j-1}, l, l^\diamond_j, l^\diamond_{j+1}, \dots, l^\diamond_{c-1})$
5   **break**
6 **return** $h^\diamond_{1,\dots,c}, l^\diamond_{1,\dots,c}$

This insertion uses a test `accept`$(l_{\text{candidate}}, l_{\text{existing}}, \tau)$ which returns true with the probability p:

$$p = \begin{cases} 1 & \text{if } l_{\text{candidate}} < l_{\text{existing}} \\ e^{-\frac{l_{\text{candidate}} - l_{\text{existing}}}{\tau}} & \text{otherwise.} \end{cases}$$

A better/lower loss value $l_{\text{candidate}}$ is always accepted. If the candidate value is worse/higher then we are the less likely to accept the worse it is. Here $\tau > 0$ is the *temperature*, gradually reduced over time by multiplying it by a constant at each step (line 18 in Algorithm 2). As the temperature τ tends towards 0, the probability of accepting a new champion with a higher/worse loss tends to 0 as well. In the extreme case of $\tau = 0$ the probability of accepting a champion with a higher/worse loss is set to 0.

Third, we update the best heuristic seen so far, $h_{\text{historic best}}$, in lines 10 through 12. We replace $h_{\text{historic best}}$ and its regularized loss $l_{\text{historic best}}$ with that of the best champion if its loss is better. Initially $l_{\text{historic best}}$ is set to ∞ in line 4.

Fourth, we form the next generation. If cumulative mutations are enabled (i.e., δ = true) then the lowest-loss s champion heuristics $h^\diamond_{1,\dots,s}$ are kept in the population in line 14. The rest of the population, the heuristics h_{s+1} through h_n, are generated as offspring of the champions $h^\diamond_{1,\dots,c}$ in line 15. Each of the offspring heuristics is created by picking a random parent among $h^\diamond_{1,\dots,c}$ and mutating it randomly with a set of mutations similar to those used by Bulitko, Hernandez, and Lelis (2021).

If cumulative mutations are not allowed (δ = false) then the new population of n heuristics are drawn randomly from the space H in line 17 and the next iteration begins. Once the budget is exhausted the best heuristic seen during synthesis, $h_{\text{historic best}}$, is returned.

By design, several existing algorithms are special cases of the algorithm above. To get the triage-enabled stochastic sampling (Bulitko, Hernandez, and Lelis 2021) we set the population size n to their triage ratio t. We set the number of champions $c = 1$ and the number of survivors $s = 0$. The temperature $\tau = 0$, there is no regularization ($\lambda = 0$) and no cumulative mutations (δ = false). To get triage-enabled simulated annealing that remembers the historic best we set $n = t, c = 1, s = 0, \tau > 0, \lambda = 0, \delta =$ true. Finally, for a basic genetic algorithm with triage and elitism the control parameters are $c \geq 1, n = c \cdot t, s = c, \tau = 0, \lambda = 0, \delta =$ true.

There are many valid combinations of the control parameters beyond the three basis configurations. We speculate that the parameters interact and thus a well performing configuration is problem-specific. Consequently we tune them via a parameter sweep on a set of problems.

5 Empirical Evaluation

We evaluated our new synthesis algorithm on video-game-style grid pathfinding (Sturtevant 2012) which is a widely used testbed for heuristic search. We chose a total of 24 maps from two games: *Dragon Age: Origins* (DAO) and *StarCraft* (SC) to represent both outdoor and indoor gaming environments. To study portability of synthesized heuristics we formed two sets -A and -B of six maps from each game. Thus we had four sets of six maps each: DAO-A, DAO-B, SC-A and SC-B.

In line with previous work, each map was treated as a rectangular grid of binary (open/obstacle) grid cells. Each cell was connected to its four cardinal neighbors and all edge costs were 1. We added a single-cell obstacle border to each map unless it was already bordered. To generate pathfinding problems on a map we computed the largest connected component of the map and then randomly placed n_{goals} distinct goals in it. For each goal we then placed n_{starts} distinct start states in the connected component.

Our loss function was defined with respect to the baseline algorithm a = A* with weighted Manhattan distance as in the previous work (Bulitko, Hernandez, and Lelis 2021).

5.1 Base Space of Heuristic Functions

We adopted a previously published (Bulitko, Hernandez, and Lelis 2021) representation of heuristics via algebraic formulae with slight modifications. Specifically we defined the heuristic space as $H = \{h(x, y, x_g, y_g) = F\}$ where (x, y) is the state for which the heuristic value is computed and (x_g, y_g) is the goal state. The formula body F is generated by the following context-free grammar:

$$\begin{aligned} F &\rightarrow T \mid U \mid B \\ T &\rightarrow x \mid x_g \mid y \mid y_g \mid \Delta x \mid \Delta y \mid C \\ U &\rightarrow \sqrt{F} \mid |F| \mid -F \mid F^2 \\ B &\rightarrow F + F \mid F - F \mid F \times F \mid \frac{F}{F} \mid \max\{F, F\} \mid \min\{F, F\} \end{aligned}$$

Here $\Delta x = |x - x_g|$, $\Delta y = |y - y_g|$ and $C \in \{1.0, 1.1, 1.2, \dots, 10.0\}$ (in the published work (Bulitko, Hernandez, and Lelis 2021) $C \in \{1, 2, \dots, 6\}$). The space includes the standard/weighted Manhattan distance $w \times (\Delta x + \Delta y)$ as it is expressible in the grammar.

5.2 Synthesis Hyperparameters

In order to choose parameters controlling our synthesis algorithm we ran a number of preliminary experiments for which we formed the problem set P_{train1} from $3 \times 3 = 9$ problems (3 goals with 3 start states for each) and P_{train2} from $10 \times 10 = 100$ problems. Each of these two training sets was generated randomly on each synthesis trial. The training set P_{train3} was fixed for all synthesis trials and consisted of $100 \times 100 = 10^4$ problems. The test set P_{test} consisted of $200 \times 200 = 4 \times 10^4$ problems, fixed for all synthesis trials.

We first set the regularizer $\lambda = 10^{-3}$ and the temperature decay factor to 0.99. Then to choose the cumulative mutation flag δ we compared the test loss of synthesized heuristics with δ = false and δ = true. The former case corresponds to the published work (Bulitko, Hernandez, and Lelis 2021) so we adapted their experimental parameters and set $s = 0, c = 1, \lambda = 0, \tau = 0, n = 20$. For each of the six DAO-A maps we then ran Algorithm 1 with 160 trials and the synthesis budget 10^8 on each trial. The test loss of the six synthesized heuristics ranged from 0.37 to 0.64 (mean 0.54) which is comparable to $[0.29, 0.62]$ (mean 0.49) reported by Bulitko, Hernandez, and Lelis (2021).* Repeating the same experiments with cumulative mutations enabled (δ = true) we obtained the test loss in the range $[0.37, 0.55]$ (mean 0.49) which suggests that cumulative mutations may have the potential to produce heuristics with higher performance in the worst case (maximum test loss of 0.55 versus 0.64). Thus we allowed accumulation of mutations henceforth.

We then moved to choose the number of survivors s, the initial temperature τ and the number of champions c by running a parameter sweep. Given the large number of configurations we ran 16 trials on each of the six maps in DAO-A and each of the six maps in SC-A. Each trial had the node-expansion budget of 10^8. Figure 1 shows the test loss averaged over the twelve maps for each parameter combination.

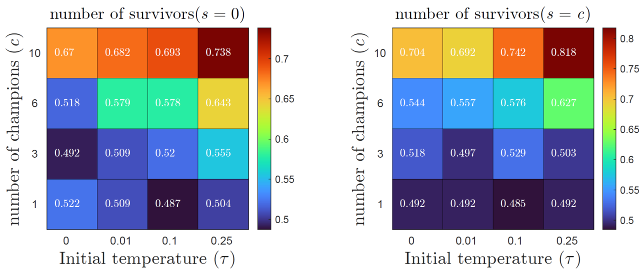

Figure 1: Test loss averaged over DAO-A and SC-A.

The results suggest that increasing the number of champions c leads to poorer synthesized heuristics. This is likely due to the fact that each champion candidate is evaluated on the larger training problem set P_{train2} (line 9 in Algorithm 2). Evaluating a heuristic on a larger problem set consumes more of the node-expansion budget for A*. Therefore, the higher the c the faster the budget is exhausted and the fewer generations of the synthesis can be run. While effects of the initial temperature τ or $s = 0$ versus $s = c$ are less pronounced we ended up selecting $s = c = 1$ and $\tau = 0.1$ for the rest of our empirical evaluation.

5.3 Synthesis with the Base Grammar

Having selected the control parameters for our algorithm, we first paralleled Bulitko, Hernandez, and Lelis (2021) and synthesized heuristics for the -A map sets. For each of the six maps from DAO-A and each of the six maps from SC-A we ran 160 synthesis trials with the parameters from Section 5.2 and the synthesis budget of 5×10^8. For each of the 12 maps we selected the single synthesized heuristic with the lowest regularized P_{train3} loss. The heuristics and their losses are listed in Tables 1 and 2.

Test loss	Heuristic
0.4007	$\mathbf{f_1} = \max\left\{\Delta x \cdot \sqrt{\frac{y_g}{x}}, \Delta y\right\}^4$
0.3627	$\mathbf{f_2} = \Delta y + 44.9 \cdot \max\{\Delta x, \Delta y\}$
0.4494	$\mathbf{f_3} = \max\left\{\frac{y_g}{(y-8.3)} \cdot \Delta y, \Delta x\right\}^2$
0.4995	$\mathbf{f_4} = \max\{100.0, \min\{y_g, \Delta y\} + y\}^2 \cdot \max\{\Delta x, \Delta y\}$
0.4894	$\mathbf{f_5} = \max\left\{\sqrt{(y + \Delta y)^2 \cdot \Delta y \cdot 11.5}, \Delta x \cdot y_g\right\}$
0.4798	$\mathbf{f_6} = \max\{\Delta y, \Delta x + \min\{\Delta x, x_g\}\}^2$

Table 1: Synthesized heuristics for maps brc202d, den000d, den501d, lak505d, orz103d, ost000a in DAO-A, manually simplified for readability.

The test loss varied between 0.27 and 0.5 which means that A* guided with these heuristics would expand 2 to 3.7 times fewer nodes than the baseline A* with the weighted Manhattan Distance for the same solution quality.

5.4 Synthesis with the Enriched Grammar

We then synthesized heuristics for the map sets DAO-B and SC-B. We compared synthesis with the base grammar to synthesis with a grammar enriched by adding heuristics previously synthesized for maps DAO-A or SC-A or both as the additional terminal symbols. As we have two A-sets of maps, we have three enriched grammars denoted by the source of their additional terminal symbols: DAO-A, SC-A and DAO-A + SC-A in Figures 2 and 3. For each of the three enriched grammars and the original base grammar, we ran three configurations of synthesis: 1 synthesis trial with the synthesis budget of 10^8 expanded nodes and 4 and 16 trials with the budget of 5×10^8. We repeated the multi-trial synthesis four times, averaged the four synthesis curves and plotted their means and standard deviations in the figures.

As expected, enhanced grammars yielded synthesized heuristics with lower losses, especially for lower synthesis budgets. Running one synthesis trial for a map with the synthesis budget of 10^8 expanded nodes with the enriched grammar DAO-A + SC-A took on average 1.6 minutes/map for DAO-B maps and 1.2 minutes/map for SC-B maps. The resulting average loss was 0.48 and 0.41 respectively which means that with the synthesized heuristics A*

*We attribute the differences to the stochastic nature of the synthesis process, randomly generated problem sets and the differences in our grammar.

Test loss	Heuristic
0.2846	$\mathbf{f_7} = \max\{\Delta y, \Delta x\} \cdot x_g + \Delta y$
0.3539	$\mathbf{f_8} = \max\{\sqrt{y} + \Delta y, \Delta x\}^2 + \Delta y$
0.4711	$\mathbf{f_9} = \Delta x + \max\{\Delta y \cdot 5.6, \Delta x\}$
0.3310	$\mathbf{f_{10}} = \max\left\{\Delta y \cdot \sqrt{\sqrt{\sqrt{\Delta y}}}, \Delta x\right\}^2 - \Delta y + \Delta x$
0.4931	$\mathbf{f_{11}} = \Delta x + \max\{1.5 \cdot \Delta y, \Delta x\} \cdot 5.6$
0.2679	$\mathbf{f_{12}} = \max\{\Delta y + \sqrt{y_g}, \Delta x\}^2$

Table 2: Synthesized heuristics for maps Legacy, Rosewood, ShroudPlatform, SpaceAtoll, Triskelion, WarpGates in SC-A, manually simplified for readability.

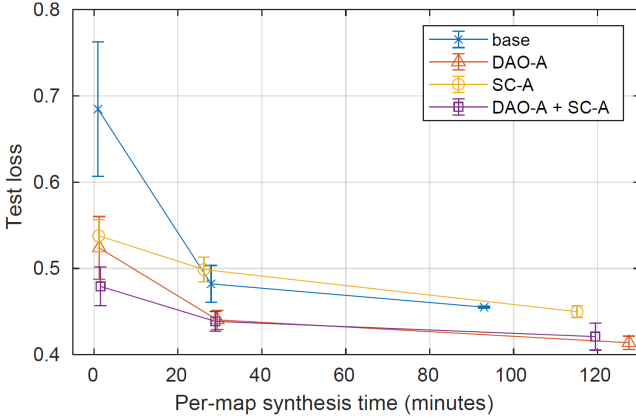

Figure 2: Synthesizing heuristics on the map set DAO-B.

produced solutions of equal quality 2.1 to 2.4 times faster than the baseline. Increasing the synthesis budget to 5×10^8 expansions and the number of trials to 16 upped the time to 119.5 and 91.7 minutes/map respectively but resulted in A* speed-up of 2.4 and 3.7 times. The graphs suggest that the test loss eventually plateaus which is likely due to limitations of our grammar as well as the preference for more compact formulae due to loss regularization.

Map	Test loss	Heuristic
brc100d	0.4850	$\mathbf{f_3}$
brc201d	0.4479	$\min\{\min\{\mathbf{f_6} - x, \mathbf{f_{10}}\}, \mathbf{f_7}\}$
den505d	0.4634	$\max\left\{\max\left\{\frac{\mathbf{f_1}}{\mathbf{f_6}}, \mathbf{f_5}\right\} + \Delta x, \mathbf{f_7}\right\}$
lak100c	0.3459	$\mathbf{f_{10}}$
orz701d	0.3943	$\mathbf{f_7}$
orz702d	0.4051	$\min\left\{\mathbf{f_3}^2, \mathbf{f_1} + y\right\}$

Table 3: Heuristics synthesized for DAO-B with the DAO-A + SC-A enriched grammar.

The best of the 16 trials (each with the budget of 5×10^8 node expansions) synthesized heuristics for -B set maps are listed in Tables 3 and 4. They were synthesized with the enriched grammar DAO-A + SC-A and thus had access to all 12 building blocks $\mathbf{f_i}$ previously synthesized on the

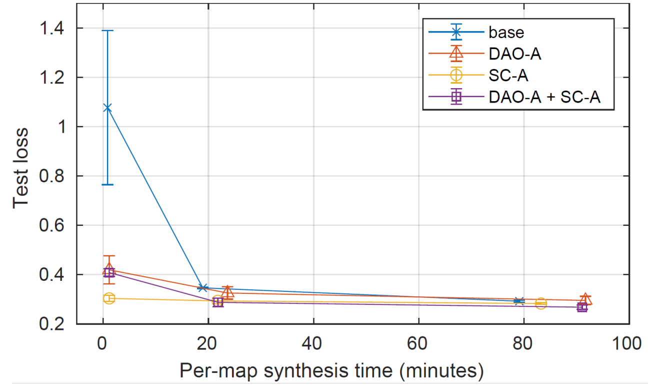

Figure 3: Synthesizing heuristics on the map set SC-B.

Map	Test loss	Heuristic
Aftershock	0.2483	$\max\left\{\frac{\mathbf{f_3}}{1.3}, \frac{\mathbf{f_5}}{\sqrt{3.1}}\right\}$
Archipelago	0.2209	$\mathbf{f_{12}}$
BigGameHunters	0.3315	$\max\{\mathbf{f_3}, \mathbf{f_5}\}$
Brushfire	0.2190	$\mathbf{f_{11}} + \Delta y$
Caldera	0.1545	$\mathbf{f_{11}} + \Delta y$
CatwalkAlley	0.4160	$\mathbf{f_9}$

Table 4: Heuristics synthesized for SC-B with the DAO-A + SC-A enriched grammar.

-A maps (Tables 1 and 2). Unsurprisingly, these heuristics synthesized with the enriched grammar are generally much shorter formulae than those synthesized with the base grammar only (Tables 1 and 2). This is likely because the building blocks $\mathbf{f_i}$ these formulae can incorporate are more powerful than the terminal symbols in the original base grammar. Note that several -B set heuristics consist of a single building block only. The fact that some heuristics originally synthesized for an A-map (e.g., $\mathbf{f_7}$ for a SC-A map Legacy) surfaced as the historic best in the synthesis for a -B map (in this case DAO-B map orz701d) promised cross-map and even cross-game portability of synthesized heuristics. We investigate such portability in the next section.

5.5 Portability of Synthesized Heuristics

A heuristic created for a given map has the potential to capture map specifics and improve A* performance on that map. At the same time, being specific to one map may negatively affect A* search on another, substantially different map. Tabular memory-based heuristics are a prime example of such specificity, lacking portability across maps. Our per-map-synthesized heuristics are algebraic formulae so *any* heuristic is applicable to *any* map. In this section we empirically study their performance degradations across maps.

We took the 12 heuristics synthesized for the -B maps (Tables 3 and 4) and computed their test loss on the 12 maps in the -B sets. We observe that portability on maps from the same game is better than portability on maps from a different game (the diagonal values in Table 5). Importantly, even

when tested on maps from a different game the synthesized heuristics outperformed the baseline search (A* with the weighted Manhattan distance) approximately $1.5 - 2$ times (test losses of 0.53 and 0.67 in the table).

Synthesis map set \ Test map set	DAO-B	SC-B
DAO-B	**0.61**	0.53
SC-B	0.67	**0.36**

Table 5: Test loss across map sets. Best values are in bold.

How much performance do we give up by testing a heuristic synthesized for a map on a different map? Table 6 shows the loss degradation values averaged across maps within the same game and a different game. Testing a heuristic synthesized for *one* of the six maps in DAO-B on *all* six DAO-B maps has the average loss degradation of 0.19 which is the price for using a heuristic on another map *within the same game*. SC-B heuristics are even more portable within the game with the average degradation of merely 0.09. As expected with both DAO-B and SC-B heuristics, their degradation was higher when used on maps from a different game.

Synthesis map set \ Test map set	DAO-B	SC-B
DAO-B	**0.19**	0.27
SC-B	0.25	**0.09**

Table 6: Degradation across map sets. Best values bolded.

We would like to draw three observations from these results. First, A*-guiding performance of a synthesized heuristic generally degrades when it is used on a map different from the map it was synthesized for. This suggests that our synthesized heuristics do capture and take advantage of map specifics. Second, the degradation is more severe across different games which suggests that maps within a game do have commonalities that our synthesized heuristics are able to exploit. Third, despite the degradation, synthesized heuristics still outperformed the baseline search even on foreign maps and foreign games.

5.6 Explainability of Synthesized Heuristics

Unlike heuristics expressed as large precomputed tables or deep neural networks, our representation via compact algebraic formulae has the potential for human readability. This is beneficial since it can help communicate synthesized heuristics and encourage their deployment for video-game pathfinding in the field. Furthermore, readable synthesized heuristics can give researchers ideas for other heuristics and algorithms. Finally, human readability can facilitate theoretical analysis such as deriving solution suboptimality bounds.

To illustrate, consider the heuristic $h(x, y, x_g, y_g) = \max\{\Delta x, \Delta y\}^2 = \max\{|x - x_g|, |y - y_g|\}^2$ whose variants have been commonly synthesized and whose wall-hugging behaviour was mentioned by Bulitko, Hernandez, and Lelis (2021). This heuristic allows A* to find solutions

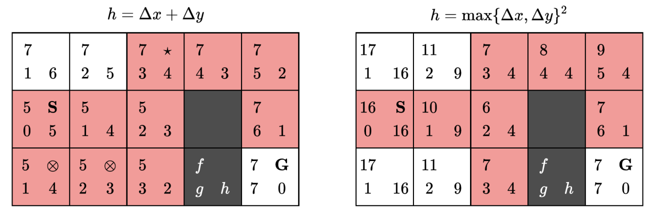

Figure 4: The synthesized $h = \max\{\Delta x, \Delta y\}^2$ causes A* to hug the wall and expand fewer states than Manhattan distance $\Delta x + \Delta y$.

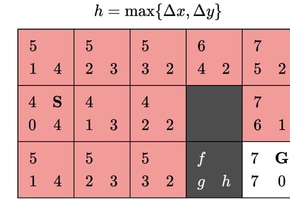

Figure 5: Removing the square from $h = \max\{\Delta x, \Delta y\}^2$.

with the average suboptimality of 1.02 while expanding on average 1767 states per problem on some 40 thousand test problems on the DAO-A map den000d. This is 3.4 times less search effort than A* with weighted Manhattan distance's interpolated 6018 states expanded on average per problem for the same average solution suboptimality.

To understand how this synthesized heuristic offers a better guidance on this map consider the toy example in Figure 4. The agent is in the cell **S** looking for a path to cell **G**. Each cell is labeled with its values of $g, h, f = g + h$. Blackened cells are walls. With h being Manhattan distance the agent expands all six cells whose f value is below $f = 7$ of the bottleneck cell marked with $\star$ (left subfigure). The bottleneck is the state that any path to the goal must pass through for this search problem. As ties among f are broken towards higher g the agent also expands four cells with $f = 7$. A total of ten cells are expanded, shaded in light red.

The synthesized heuristic $h = \max\{\Delta x, \Delta y\}^2$ ignores the smaller of the Δx and Δy terms (instead of adding them together as Manhattan distance does) and reduces the h value non-linearly along a path. Guided by it, A* goes straight east until it hits the wall (right subfigure). Hugging the wall, the search will then expand to the goal. A total of eight cells are expanded, shaded in light red.

In this toy example only two state expansions are saved. Interestingly, these two cells ($\otimes$ in the figure) would be expanded by weighted A* for *any* value of the weight since their g values 1 and 2 are below $g = 3$ of the bottleneck $\star$ and their h values $3, 4$ do not exceed that of the bottleneck (4). Also note that removing the square from the synthesized heuristic actually *increases* state expansions to 12 (Figure 5).

In practice the wall-hugging behaviour induced by the synthesized heuristic can save many more expansions on more realistic pathfinding problems. To illustrate, consider the problem with the start state $(x = 19, y = 177)$ and the goal state $(x = 410, y = 162)$ on the map den000d. Its op-

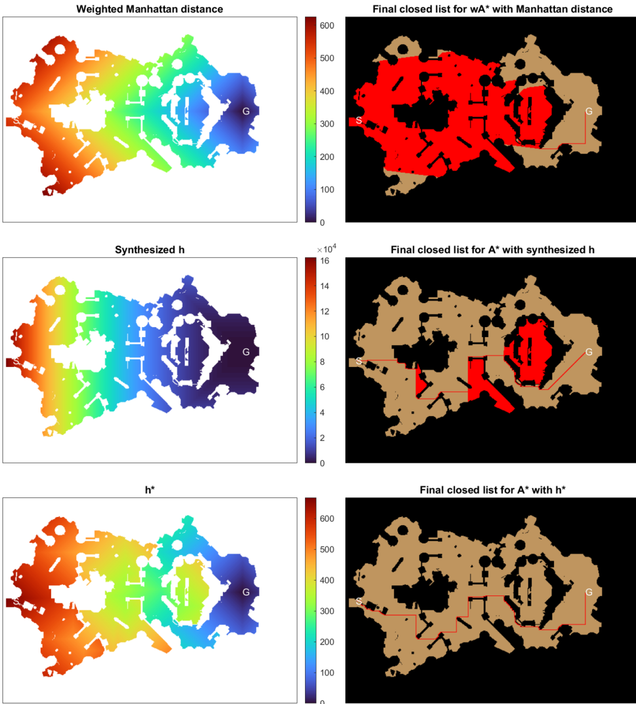

Figure 6: Comparing heuristic values and final closed lists of the baseline (top row), A* with the synthesized heuristic (middle row) and A* with the perfect heuristic h^* (bottom row). Final closed lists are shown in red. The white S and G denote the start and goal. The weight $w = 1.2969$ for Manhattan distance was picked to match average suboptimality of the synthesized heuristic.

timal solution has 652 steps (bottom right plot in Figure 6). Visually, weighted Manhattan distance $h = w(\Delta x + \Delta y)$ matches the perfect heuristic h^* much better than the synthesized heuristic (plots in the left column). However, weighted Manhattan distance does not account for the obstacles and combines Δx and Δy linearly. Consequently, it causes A* to expand a large number of states throughout the map as seen in its final closed list (top right plot). As a result, on this problem A* with weighted Manhattan distance expands 39799 states to find a solution of 666 steps ($\alpha = 1.021$).

The synthesized heuristic $h = \max\{\Delta x, \Delta y\}^2$ also does not account for the map's obstacles but it does induce a wall-hugging behaviour in A* that saves many state expansions. Indeed, on this problem A* expands 4.6 times fewer states (middle right plot), yielding a solution of 668 steps ($\alpha = 1.025$) found with only 8608 state expansions.

By not accounting for specific obstacles on the map (unlike a pre-computed memory-based heuristic) the synthesized heuristic causes A* to explore several dead-end chambers closer to the goal which a perfect heuristic would avoid (bottom right in the figure). However, a memory-based heuristic pre-computed for this map would not work on another map whereas the formula-based $h = \max\{\Delta x, \Delta y\}^2$ induces the wall-hugging behaviour useful on *many* maps.

Another advantage of automatically synthesized algebraic heuristic functions is that they can allow for a simple bounded-suboptimality analysis. For instance, solutions A* finds while guided by the synthesized heuristic function $h(s) = \Delta y + \max\{6.6\Delta y, 7.7\Delta x\}$ are guaranteed to be no costlier than 7.7 times the optimal solution cost because $h(s) = \Delta y + \max\{6.6\Delta y, 7.7\Delta x\} \leq \Delta y + 6.6\Delta y + 7.7\Delta x \leq 7.7(\Delta y + \Delta x) \leq 7.7h^*(s)$.

Another synthesized heuristic $h(s) = \max\{\Delta y, \Delta x\} x_g + \Delta y$ guarantees solution suboptimality no worse than $x_g + 1$ times optimal because $h(s) = \max\{\Delta y, \Delta x\} \cdot x_g + \Delta y \leq (\Delta y + \Delta x)x_g + \Delta y \leq \Delta y(x_g + 1) + \Delta x \cdot x_g \leq (x_g + 1)(\Delta x + \Delta y) \leq (x_g + 1)h^*(s)$.

Such bounds are more difficult to derive for less transparent heuristic representations such as deep neural networks.

6 Open Questions & Future Work

While we used video-game-map pathfinding as the testbed, future work will consider other search spaces such as combinatorial puzzles and general planning (Yoon, Fern, and Givan 2008). Furthermore, in this paper we synthesized heuristics for a given search graph (i.e., per a video-game map). Future work will synthesize heuristics for multiple search graphs. Will increasing the number and diversity of graphs for which a single heuristic formula is synthesized cause the synthesis to converge to a one-size-fits-all heuristic?

Another direction for future work is to jointly synthesize heuristic functions and heuristic search algorithms that use them. In particular, one can synthesize the priority function used to order the Open list in A* and see if the joint synthesis can outperform synthesizing only a heuristic or only a priority function. One can compare synthesized priority functions to human-designed ones (Chen and Sturtevant 2021).

As the synthesis space is made larger it will be even more important to evaluate candidate heuristics efficiently. In this paper we used a series of progressively larger problem sets to discard poor heuristics quickly. An alternative is to use a surrogate fitness function which does not involve running A* with a heuristic at all. Recent work (Bulitko and Botea 2021) used convolutional networks as a proxy fitness function in the evolution of crosswords puzzles. Another possibility is to use an error between a candidate formula-based heuristic and the perfect table-based heuristic h^* as the loss function. While Section 5.6 shows that such errors may not align with the resulting A* performance, the error may be used as a *part* of the guidance of the synthesis process.

Future work will further investigate sensitivity of synthesized formulae performance to hyperparameter values as well as advanced automated tuning of the hyperparameters. It will also be of interest to consider optimizations in evaluating the formulae representing heuristics as well as automatic simplification of such formulae. Finally, by adding constraints to the synthesis process and/or modifying the loss function one can attempt to synthesize combinations of heuristic functions and heuristic algorithms that produce solutions with better suboptimality bounds.

7 Conclusions

Traditionally heuristic functions for video-game pathfinding have been either generic and manually constructed (e.g., the Manhattan distance) and lacking A*-guiding performance or pre-computed for each map and thus lacking portability and human readability. Recent work (Bulitko, Hernandez, and Lelis 2021) attempted to combine advantages of both by automatically synthesizing heuristics represented by simple algebraic formulae. Since they are synthesized for each map, there is a potential for higher performance by capturing some specifics of the map.

While promising, that study left a number of important questions which we attempted to answer here. First, we updated their synthesis algorithm with cumulative mutations, stochastic acceptance and regularization which allowed us to investigate performance impact of the parameter combinations. Second, we showed that the synthesized heuristics appear to capture and take advantage of specifics of individual maps within a game and a game as a whole. While there is a degradation in performance when a heuristic synthesized for a map is applied to a different map or even a different game, on average the resulting performance is still better than the baseline search. This is promising for game developers since it would spare them from having to synthesize a different heuristic for every map and thus would allow to use pre-synthesized heuristics even on user-made maps. Finally, the simple formula-based representation of our synthesized heuristics can enable human readability and analysis. To illustrate, we showed how a commonly synthesized compact heuristic induces wall-hugging behavior in A* and consequently substantially speeds it up. We also demonstrated how simplicity of synthesized formula-based heuristics can allow to prove suboptimality bounds.

Acknowledgments

We appreciate financial support from NSERC. We also appreciate support from Compute Canada and consultation from Jonathan Schaeffer. This research was partially funded by Canada's CIFAR AI Chairs program.

References

Adler, D. 1993. Genetic algorithms and simulated annealing: A marriage proposal. In *Proceedings of the IEEE International Conference on Neural Networks*, 1104–1109.

Albarghouthi, A.; Gulwani, S.; and Kincaid, Z. 2013. Recursive program synthesis. In *Proceedings of the International Conference on Computer-aided Verification*, 934–950.

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *Proceedings of Formal Methods in Computer-Aided Design*, 1–8. (a tutorial).

Björnsson, Y.; and Halldórsson, K. 2006. Improved Heuristics for Optimal Path-finding on Game Maps. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 9–14.

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

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

Bulitko, V.; and Botea, A. 2021. Evolving Romanian Crossword Puzzles with Deep Learning and Heuristic Search. In *Proceedings of the IEEE Conference on Games (CoG)*.

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

Chen, J.; and Sturtevant, N. R. 2021. Necessary and Sufficient Conditions for Avoiding Reopenings in Best First Suboptimal Search with General Bounding Functions. In *Proceedings of the AAAI Conference on AI*, 3688–3696.

Culberson, J.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proceedings of the Conference on Planning*, 13–24.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *Journal of Artificial Intelligence Research*, 30: 213–247.

Hart, P.; Nilsson, N.; and 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.

Hernandez, S. P.; and Bulitko, V. 2021. Speeding Up Heuristic Function Generation via Automatically Extending the Formula Grammar. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science*, 220(4598): 671–680.

Korf, R.; and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1–2): 9–22.

Korf, R. E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 700–705.

Odena, A.; Shi, K.; Bieber, D.; Singh, R.; Sutton, C.; and Dai, H. 2020. BUSTLE: Bottom-Up program synthesis through learning-guided exploration. In *Proceedings of the International Conference on Learning Representations*.

Shi, K.; Bieber, D.; and Singh, R. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. In *Proceedings of the NeurIPS Workshop on Computer-Assisted Programming*.

Shi, K.; Dai, H.; Ellis, K.; and Sutton, C. 2021. CrossBeam: Learning to Search in Bottom-Up Program Synthesis. In *Proceedings of the International Conference on Learning Representations*.

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

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-Based Heuristics for Explicit State Spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 609–614.

Udupa, A.; Raghavan, A.; Deshmukh, J. V.; Mador-Haim, S.; Martin, M. M.; and Alur, R. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. *SIGPLAN Not.*, 48(6): 287–296.

Yang, X. 2011. Metaheuristic Optimization. *Scholarpedia*, 6(8): 11472. Revision #91488.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *Journal of Machine Learning Research*, 9(4).

Zgierski, J. R. 1993. *On stochastic sorting.* Ph.D. thesis, Carleton University.