Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding

Vadim Bulitko
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
bulitko@ualberta.ca

Sergio Poo Hernandez
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
pooherna@ualberta.ca

Levi H. S. Lelis
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
levi.lelis@ualberta.ca

***Abstract*—Heuristic search is widely used in games for pathfinding and general planning. High-quality heuristic functions are key to finding a low-cost solution quickly. Commonly used heuristic functions for video-game pathfinding are either manually designed and generic or pre-computed for a specific map. The former fail to take advantage of pathfinding specifics while the latter tend to have a large memory footprint, may require substantial pre-computation and are not portable to other maps or easily presentable to humans. In this work we attempt to combine the best of both approaches by automatically synthesizing well performing pathfinding-specific yet compact and human-readable heuristics. We do so by defining a space of algebraic formulae expressing heuristic functions and then conducting an automated search of the space. To make the synthesis tractable we employ a multi-tier evaluation which allows us to quickly filter out low-quality heuristics while saving time to more thoroughly evaluate better ones. Such triage of candidate heuristics enables us to synthesize compact heuristics that outperform the standard baseline on video-game pathfinding benchmarks. By then adding the synthesized heuristics back to the synthesis space we show that synthesis on new maps can be substantially sped up to merely few minutes per map.**

***Index Terms*—heuristic search, heuristic function, program synthesis, video-game pathfinding**

I. Introduction

Heuristic search has been widely used in video games for pathfinding [1]. Its performance crucially depends on a heuristic function (a *heuristic* for short) which guides the search. Commonly used heuristics are either generic (e.g., Manhattan distance) or pre-computed for a given map [2]. The former are easy to explain as they are expressed by compact intuitive formulae but do not take advantage of domain specifics and offer basic performance. The latter achieve a greater performance but need to be pre-computed for each map, can have large memory requirements and cannot be compactly presented to humans.

In this paper we attempt to combine the best of both types by automatically discovering formula-based heuristics via program synthesis. Similar to Manhattan distance, our synthesized heuristics tend to be compact, human-readable formulae. They are portable across different maps, easily human-readable and have negligible memory requirements. Furthermore, by exploiting particulars of video-game pathfinding our machine-synthesized formulae substantially outperform a common baseline (weighted Manhattan distance).

Fundamentally we propose that human designers define a *space of heuristic functions* instead of designing individual heuristics. Then machines can search through this space, finding well performing heuristics tailored to a given class of search problems (e.g., pathfinding problems on maps from a particular video game).

This approach has several advantages over manual design. **First**, it enables an automatic synthesis of heuristic functions tailored to even a small set of problems (e.g., pathfinding on a single map) in an attempt to exploit peculiarities/patterns shared by such problems. **Second**, the space of heuristic functions can use a human-readable representation which makes their adoption by a game developer more likely. To illustrate, one of our synthesized heuristics estimated the remaining distance to goal by non-linearly combining the remaining horizontal and vertical distances: $h = \max\{\Delta x, \Delta y\}^2$. In doing so it induced a wall-hugging behavior in A* thereby substantially reducing the number of state expansions relative to the baseline weighted Manhattan distance. Yet that formula can be written on a napkin and implemented in a video game as a drop-in replacement for Manhattan distance which is substantially easier than implementing a pattern database [3], a memory-based heuristic [2] or a Euclidean embedding [4]. **Third**, the machine-synthesized formulae can be analyzed by humans to give tradition-defying insights into heuristic design. Such insights can then inform design of the next space of heuristics for an iterative computer-assisted design process.

The critical points in realizing these advantages are to keep the space of heuristic functions tractably small while using a fast synthesis method over that space. This paper makes contributions to address both points. We adopt the recently proposed human-readable representation of heuristic functions in pathfinding as algebraic formulae encoded by syntax trees [5]. The space of heuristic functions is then defined by a context-free grammar. In the spirit of a recent extension [6], we enrich the grammar by adding synthesized heuristics back to it. Unlike the recent work [6], we add all synthesized heuristics as atomic terminal nodes without any de-composition or filtering. Additionally, unlike the referenced work [5], [6], we evaluate the approach in the more com-

978-1-6654-3886-5/21/$31.00

mon, non-real-time heuristic search setting. To keep synthesis tractable we propose triage-based sampling designed to scale with computational resources. Consequently, our synthesis can run on home computers and on large clusters.

II. Problem Formulation

In this paper we consider the problem of synthesizing heuristic functions for the standard A*-based heuristic search. We aim to develop a synthesis method that minimizes human engineering, is easy to implement and produces human-readable heuristics that have a negligible memory footprint. Such synthesized heuristics should take advantage of particulars of the search domain (e.g., be pathfinding-specific) and yet be portable (e.g., heuristics synthesized for one map should work well on a similar map).

We formalize the synthesis problem as an optimization problem of minimizing the *loss* $\ell(a, h, P)$ where a is a search algorithm guided by a heuristic h and P is a set of search problems. Our loss ℓ is the ratio of mean search effort spent by a with h solving problems in P to the mean search effort of a baseline algorithm and a baseline heuristic, measured for a matching average solution quality. For instance, $\ell = 0.5$ means that a with h finds solutions of the same average quality as the baseline while doing half the search effort on average. In doing so ℓ combines two measures, search effort and the resulting solution quality, into a single scalar.

III. Related Work

Memory-based heuristics have been developed for various search spaces [2], [3], [7], [8]. They provide effective guidance on a class of search problems (e.g., pathfinding on a particular map). They do, however, tend to have a considerable memory footprint, lack portability (e.g., cannot be used as-is on another map) and can be difficult for humans to read.

Guidance offered by a heuristic function can often be improved by computing it with respect to a closer subgoal instead of the original distant goal. Such subgoals can be automatically placed on a map [1], [9], [10]. These methods are complementary to our approach as our synthesized heuristics can also be computed with respect to a subgoal.

Another way to compute high-performance heuristics is to embed the structure of the original search graph in a Euclidean space [4], [11]. Then Euclidean distance between embedded vertices can serve as a heuristic. The embedding procedure is non-trivial and needs to run for each map. Yet another body of work learns heuristic functions [12]–[15]. Such methods represent learned heuristic functions as neural networks which can make human readability a challenge. Learned policy gradient uses differentiable optimization to automatically discover reinforcement-learning concepts such as value functions [16]. We seek to discover heuristic functions in a compact and human-readable representation.

Procedural content generation (PCG) is an active field in video games with various types of game content successfully generated in both research [17]–[20] and in the field [21]–[24]. Yet there has been no published PCG work on synthesizing heuristic functions expressed in a compact human-readable way beyond the work we build on [5], [6].

IV. Our Approach

In line with previous work [5], [6], we let synthesized heuristics take advantage of specifics of a given map by synthesizing heuristics on a *per-map* basis. The downside of this approach is a substantial per-map synthesis cost. The primary contribution of this work is making per-map synthesis practical by using a rich, procedurally extended grammar that defines the space of heuristics. As a result, we are able to synthesize an effective map-specific heuristic in a few minutes.

Our synthesis method is intentionally simple which (i) allows for an easier deployment at a video-game studio and (ii) suggests even better results with a more complex optimization approach (e.g., genetic algorithms). In our method, we sample heuristics from a set H of heuristics and use a series of progressively larger *training sets* of search problems to evaluate the samples. The intuition is simple: a candidate heuristic h drawn randomly from H is likely to yield low performance. We want to be able to perform a *triage* and filter out such heuristics quickly without wasting time on computing just how bad their performance is. Thus, given an h randomly drawn from H in line 5 in Algorithm 1 we first run the algorithm a with h on a small training set of problems P_{train1} (line 6). We do so n times and select a heuristic h_1 with the lowest loss on P_{train1} in the loop in lines 4 – 9.

Algorithm 1: A single synthesis trial

input : training problem sets $P_{\text{train1}}, P_{\text{train2}}$, heuristic space H, synthesis budget b, loss function ℓ, triage ratio n
output: synthesized heuristic h_{trial}

```
1  l_2 ← ∞
2  repeat
3      l_1 ← ∞
4      for k ∈ {1, ..., n} do
5          draw h ~ H
6          l ← ℓ(a, h, P_train1)
7          if l < l_1 then
8              l_1 ← l
9              h_1 ← h
10     l ← ℓ(a, h_1, P_train2)
11     if l < l_2 then
12         l_2 ← l
13         h_trial ← h
14 until b is exhausted
```

If the set P_{train1} is small (i.e., $|P_{\text{train1}}| \ll |P|$) then computing the loss $\ell(a, h, P_{\text{train1}})$ is fast but the result may not be representative of the loss on all problems P of interest. To alleviate such overfitting to P_{train1} we test the selected h_1 on a larger training set P_{train2} in line 10. If the resulting loss is lower than the previously observed best then we update the running best in line 13.

Computing heuristic loss on sets P_{train1} and P_{train2} involves running the search algorithm a which expands states. The total number of states expanded counts against the synthesis budget b. Once the budget is exhausted (line 14) the lowest P_{train2}-loss heuristic found becomes the trial's output, h_{trial}.

The single synthesis trial described in Algorithm 1 produces a single heuristic h_{trial}. Since it is based on random sampling from the space H, the resulting heuristic is likely to vary significantly from trial to trial. In response we run many synthesis trials independently. The h_{trial} synthesized on each trial is evaluated on the third and largest training problem set P_{train3}. The heuristic with the lowest P_{train3}-loss becomes the single output of the entire synthesis process, h_{synth}. The synthesized heuristic h_{synth} is thus our solution to the optimization problem. Its *test loss* $\ell(a, h_{\text{synth}}, P_{\text{test}})$ is computed on a separate set P_{test}.

Note that the synthesis process is designed to take advantage of both within-CPU parallelism and across-CPU parallelism. Specifically, we can compute the loss function $\ell(a, h, P)$ by partitioning the problem set P by its goal states. Then each CPU core computes ℓ for all start states for a single goal in P. Our across-CPU parallelization is realized by running each synthesis trial on a separate CPU (e.g., a cluster node).

V. Empirical Evaluation

We used video-game maps from the standard benchmark repository MovingAI [25]. Two non-overlapping sets (A and B) of six *Dragon Age: Origins* maps each were chosen to cover diverse pathfinding scenarios such as open outdoor pathfinding versus cluttered indoor pathfinding (Figures 1 and 3). We treated the maps as four-connected grids with all edge/move costs being 1. We added a single-cell border to each map if it was not already bordered.

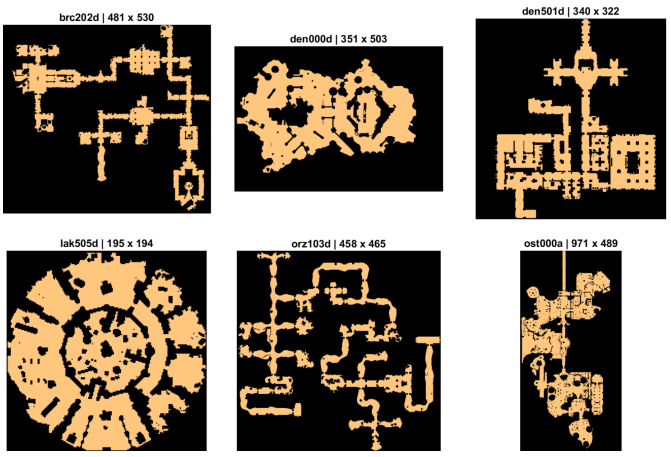

Fig. 1. Map set A.

To generate a set of problems on a map we first computed the largest connected component of the map, randomly selected N distinct goal states in it and, for each of them, randomly selected M start states. This gave us $N \times M$ problems on a map. Since each goal state is shared by M problems a heuristic function has to be computed only N times for a set of $N \times M$ problems.

A. Baseline Space of Heuristic Functions

We adopted a previously published [5] representation of heuristics via algebraic formulae. 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:

$$
\begin{aligned}
F &\rightarrow T \mid U \mid B \\
T &\rightarrow x \mid x_g \mid y \mid y_g \mid \Delta x \mid \Delta y \mid C \\
C &\rightarrow 1 \mid 2 \mid \ldots \mid 6 \\
U &\rightarrow \sqrt{F} \mid |F| \mid -F \mid F^2 \\
B &\rightarrow F + F \mid F - F \mid F \times F \mid \frac{F}{F} \mid \max\{F, F\} \mid \min\{F, F\}
\end{aligned}
$$

Here $\Delta x = |x - x_g|$ and $\Delta y = |y - y_g|$. The space includes Manhattan distance $\Delta x + \Delta y$ and weighted Manhattan distance $w \times (\Delta x + \Delta y)$ as both are expressible in the grammar.

B. Synthesis Method

Using maps from the MovingAI repository as well as our own synthetic maps [6], we explored the space of hyperparameters for the synthesis process and chose the following: per-map training sets are P_{train1} of 9 problems (3 goals $\times$ 3 start states for each), P_{train2} of 100 problems (10 goals $\times$ 10 starts) and P_{train3} of $100 \times 100 = 10^4$ problems. The sets P_{train1} and P_{train2} were generated randomly on each trial. The set P_{train3} was generated randomly once per map and kept constant for all trials so that the P_{train3}-losses from different trials were meaningfully comparable. Finally, for each map we generated a single random test set of problems P_{test} which contained $200 \times 200 = 4 \times 10^4$ problems. The triage ratio n in Algorithm 1 was set to 20.

We used the classic A* that does not re-open nodes on the closed list as our algorithm a. Ties in $f = g + h$ were broken towards higher g. Remaining ties were broken in an arbitrary fixed order. Thus all loss values were computed as $\ell(\text{A*}, h, P)$.

Our synthesized heuristics can be inadmissible causing A* to produce a suboptimal solution. Thus the baseline had to be a suboptimal algorithm as well. We chose the classic weighted A* [26] without re-opening closed nodes. It used the priority function $f = g + w \times h$ with $w \in \{1, 2, \ldots, 10\}$ and Manhattan distance as the heuristic. We linearly interpolated between the ten points (Figure 2). Higher weights tend to yield longer paths but fewer states expanded to find them. We computed two sets of the baseline data for each map: one using problems in the training set P_{train3} and one with the test problems P_{test}. The former baseline data was used to compute loss during synthesis while the latter was used to compute test loss.

C. Heuristic Synthesis on Map Set A

We ran 160 synthesis trials for each of the six maps in set A (Figure 1) with the synthesis budget $b = 10^8$ states. Collectively, 160 trials took an average of 9.6 hours per map. Evaluating outcomes of each trial on the training set P_{train3} took additional 10.4 hours per map. For each map the

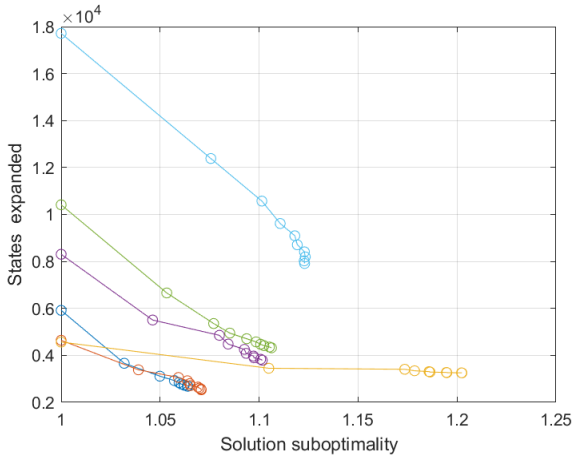

Fig. 2. Weighted A* as the baseline. The six lines correspond to the six maps in map set B. Markers are per-map averages over the 4×10^4 test problems. Each marker corresponds to a value of weight $w \in \{1, 2, \ldots, 10\}$.

synthesized heuristic with the lowest P_{train3}-loss is listed in the Table I, manually simplified for clarity.

Each of the six heuristics was then evaluated on the test set P_{test}. The test loss varied between 0.29 and 0.62 (Table I). Thus our synthesized heuristics *sped up* weighted A* search by 1.6 to 3.4 times without sacrificing solution quality.

TABLE I
HEURISTICS FOR MAP SET A SYNTHESIZED WITH BASE GRAMMAR.

Map	Test loss	Synthesized heuristic
`brc202d`	0.4990	$\mathbf{h_1} = \max\{\Delta y^2, \Delta x \times y_g\}^2$
`den000d`	0.2907	$\mathbf{h_2} = 75 \times \max\{\Delta y, \Delta x\}$
`den501d`	0.6232	$\mathbf{h_3} = \max\{\sqrt{6} + \Delta x, \Delta y\}^2$
`lak505d`	0.5771	$\mathbf{h_4} = \max\{\Delta y, \Delta x\}^4$
`orz103d`	0.5422	$\mathbf{h_5} = \max\{\Delta y, \Delta x\}^4$
`ost000a`	0.4100	$\mathbf{h_6} = \max\{\Delta y, \Delta x\}^4 - \sqrt{1 - \Delta y}$

D. Heuristic Synthesis on Map Set B

The synthesized heuristics are effective in part because they contain important building blocks. For instance, the expression $\max\{\Delta y, \Delta x\}^2$ causes A* to hug walls in its search which is efficient in video-game pathfinding.

To measure portability of the heuristics synthesized for maps in set A we applied each heuristic to each of the novel six maps in map set B (Figure 3). Per-map average loss was $0.6030, 0.5063, 0.7017, 0.4811, 0.4363, 0.4548$ for maps `brc100d`, `brc201d`, `den505d`, `lak100c`, `orz701d`, `orz702d` respectively. Further averaged over the six maps the test loss was 0.5305 (i.e., a 1.9x speed-up over the baseline).

Is it possible to achieve even better performance on set-B maps *without* having to re-discover useful building blocks in the synthesized heuristics? To answer this question, we added the six set-A heuristics (Table I) as six additional terminal symbols by modifying the grammar as follows:

$$T \to x \mid x_g \mid y \mid y_g \mid \Delta x \mid \Delta y \mid C \mid \mathbf{h_1} \mid \mathbf{h_2} \mid \mathbf{h_3} \mid \mathbf{h_4} \mid \mathbf{h_5} \mid \mathbf{h_6}$$

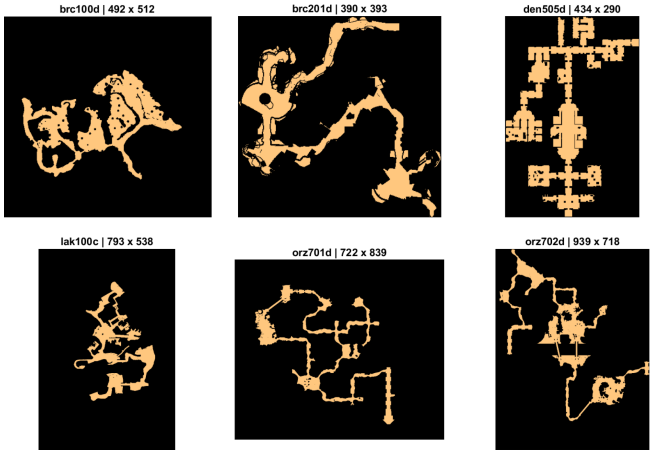

Fig. 3. Map set B.

where each $\mathbf{h_i}$ takes (x, y, x_g, y_g) as the input.

The *enriched grammar* was then used to synthesize six new heuristics for map set B. To do so we ran a single synthesis trial per map with the synthesis budget of 10^8 states. The synthesized heuristics for each of set-B maps are found in Table II (manually simplified for clarity). Despite the fact that the synthesis took an average of only 9.3 minutes per map, the synthesized heuristics have the test loss of 0.4027 to 0.6670, causing A* to expand 1.5 to 2.5 times fewer states than the baseline to produce paths of equal quality. Note that the synthesized heuristics all incorporate the new terminal nodes $\mathbf{h_i}$ in the enriched grammar. Averaged over the six maps the test loss is 0.4975 which is better than simply using set-A heuristics on set-B maps (average loss of 0.5305).

We then conducted a scalability study by running a series of progressively costlier synthesis trials with the base and the enriched grammars. For the base-grammar runs we used the synthesis budget of 10^8 with the number of synthesis trials in $\{1, 4, 16, 64, 160\}$. We then increased the budget range to $5 \cdot 10^8$ and ran 160 more trials. For the synthesis runs with the enriched grammar we used the synthesis budget of 10^8 and the number of synthesis trials in $\{1, 4, 16, 64\}$.

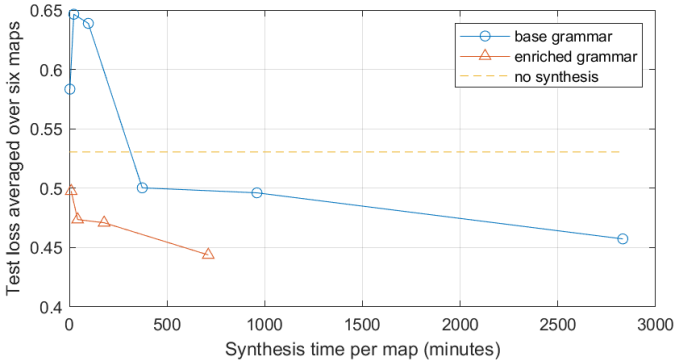

Fig. 4. Heuristics synthesized for set-B maps using different grammars.

The results are plotted in Figure 4. Heuristics synthesized specifically for set-B maps indeed outperformed set-A

TABLE II
HEURISTICS FOR MAP SET B SYNTHESIZED WITH ENRICHED GRAMMAR.

Map	Test loss	Synthesized heuristic
brc100d	0.5741	$\mathbf{h_6} + \Delta x - \sqrt{y_2}$
brc201d	0.4602	$(\mathbf{h_2} + 5)^2 + \Delta x - y$
den505d	0.6670	$\frac{1}{x} \times \left(\Delta x^2 + \mathbf{h_2}\right)^2$
lak100c	0.4027	$\min\left\{\Delta y, \min\left\{4, 1 - \max\left\{\Delta y, y\right\}\right\}^2\right\} + \max\left\{\mathbf{h_4}, \mathbf{h_3}, \mathbf{h_6}\right\}$
orz701d	0.4080	$\Delta y - y + \sqrt{\max\left\{{\mathbf{h_6}}^2, \sqrt{\mathbf{h_3}}\right\}} + \mathbf{h_6}$
orz702d	0.4727	$\max\left\{\frac{\mathbf{h_1}}{4}, 3 - \mathbf{h_1}\right\}$

heuristics (shown with the flat dashed line the figure) with either grammar. However, the enriched grammar substantially accelerates the synthesis. For instance, in about 9 minutes/map synthesis with the enriched grammar yielded the mean test loss of 0.4975 whereas it took about 16 hours/map with the base grammar to do better.

VI. FUTURE WORK

In this paper we used a series of progressively larger problem sets to discard poor heuristics quickly. One alternative is to use a surrogate fitness function which does not involve running A* at all (e.g., a deep neural network). Future work will also explore more advanced synthesis methods such as evolutionary computing [5]. Another important direction for future work is to explicitly bias synthesis towards more compact and therefore likely more human-readable heuristics. Finally, future work will attempt to jointly synthesize heuristics and search algorithms [27].

VII. CONCLUSIONS

Heuristic search is widely used in video games for pathfinding and general planning. Its performance depends critically on the guiding power of a heuristic function. Traditionally used heuristics are either manually constructed but generic or precomputed on a per-map basis and require substantial memory. In this paper we kept the compact representation and portability of the former while increasing performance towards the latter. To do so we conducted a stochastic search of a space of heuristic functions expressed as algebraic formulae, evaluating candidate heuristics on a set of progressively larger training sets. To make such per-map synthesis practical we procedurally extended the grammar that defines the synthesis space. As a result, we were able to synthesize effective and compact heuristics in less than ten minutes per map.

ACKNOWLEDGMENT

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

REFERENCES

[1] N. R. Sturtevant, D. Sigurdson, B. Taylor, and T. Gibson, "Pathfinding and abstraction with dynamic terrain costs," in *AIIDE*, 2019, pp. 80–86.
[2] N. R. Sturtevant, A. Felner, M. Barrer, J. Schaeffer, and N. Burch, "Memory-based heuristics for explicit state spaces," in *IJCAI*, 2009, pp. 609–614.
[3] A. Felner, R. E. Korf, R. Meshulam, and R. C. Holte, "Compressed pattern databases," *JAIR*, vol. 30, pp. 213–247, 2007.
[4] L. Cohen, T. Uras, S. Jahangiri, A. Arunasalam, S. Koenig, and T. S. Kumar, "The FastMap algorithm for shortest path computations," in *IJCAI*, 2018, pp. 1427–1433.
[5] V. Bulitko, "Evolving initial heuristic functions for agent-centered heuristic search," in *COG*, 2020, pp. 534–541.
[6] S. P. Hernandez and V. Bulitko, "Speeding up heuristic function generation via automatically extending the formula grammar," in *SoCS*, 2021.
[7] J. Culberson and J. Schaeffer, "Pattern Databases," *Computational Intelligence*, vol. 14, no. 3, pp. 318–334, 1998.
[8] Y. Björnsson and K. Halldórsson, "Improved heuristics for optimal pathfinding on game maps," in *AIIDE*, 2006, pp. 9–14.
[9] N. R. Sturtevant, "Memory-efficient abstractions for pathfinding," in *AIIDE*, 2007, pp. 31–36.
[10] R. Lawrence and V. Bulitko, "Database-driven real-time heuristic search in video-game pathfinding," *IEEE TCIAIG*, vol. 5, no. 3, pp. 227–241, 2013.
[11] D. C. F. Rayner, M. H. Bowling, and N. R. Sturtevant, "Euclidean heuristic optimization," in *AAAI*, 2011, pp. 81–86.
[12] M. Samadi, A. Felner, and J. Schaeffer, "Learning from multiple heuristics," in *AAAI*, 2008, pp. 357–362.
[13] L. H. Lelis, R. Stern, S. Jabbari Arfaee, S. Zilles, A. Felner, and R. C. Holte, "Predicting optimal solution costs with bidirectional stratified sampling in regular search spaces," *AIJ*, vol. 230, pp. 51–73, 2016.
[14] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the Rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, 07 2019.
[15] L. Orseau and L. H. S. Lelis, "Policy-guided heuristic search with guarantees," in *AAAI*, 2021.
[16] J. Oh, M. Hessel, W. M. Czarnecki, Z. Xu, H. P. van Hasselt, S. Singh, and D. Silver, "Discovering reinforcement learning algorithms," in *NeurIPS*, vol. 33, 2020, pp. 1060–1070.
[17] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural content generation: Goals, challenges and actionable steps," in *Dagstuhl Follow-Ups*, vol. 6, 2013.
[18] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *TCIAIG*, vol. 9, no. 1, pp. 25–41, 2017.
[19] V. Bulitko, S. Carleton, D. Cormier, D. Sigurdson, and J. Simpson, "Towards positively surprising non-player characters in video games," in *EXAG/AIIDE*, 2017, pp. 34–40.
[20] V. Bulitko, M. Walters, M. Cselinacz, and M. Brown, "Evolving NPC behaviours in A-life with player proxies," in *EXAG/AIIDE*, 2018.
[21] S. Risi, J. Lehman, D. B. D'Ambrosio, R. Hall, and K. O. Stanley, "Combining search-based procedural content generation and social gaming in the petalz video game." in *AIIDE*, 2012.
[22] Xbox Wire Staff, "Forza horizon 2: What's a drivatar, and why should I care?" *Xbox Wire*, 2014.
[23] Hello Games, "No Man's Sky Next," 2018.
[24] T. Soule, S. Heck, T. E. Haynes, N. Wood, and B. D. Robison, "Darwin's Demons: Does evolution improve the game?" in *European Conference on the Applications of Evolutionary Computation*, 2017, pp. 435 – 451.
[25] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *TCIAIG*, vol. 4, no. 2, pp. 144 – 148, 2012.
[26] I. Pohl, "Heuristic search viewed as path finding in a graph," *AIJ*, vol. 1, no. 3, pp. 193 – 204, 1970.
[27] V. Bulitko, "Evolving real-time heuristic search algorithms," in *ALIFE*, 2016, pp. 108–115.