Synthesizing Heuristics for A*

Jeremy Costello
jeremy1@ualberta.ca
Department of Electrical and Computer Engineering
University of Alberta

Abstract

Guided bottom-up search is used to synthesize heuristics for navigating video game maps. The background of the A* and guided bottom-up search algorithms is provided. The specific implementations of these algorithms used for experiments are explained, including a vectorized implementation of A* and novel metrics for guiding bottom-up search. The algorithms are used to synthesize heuristics for three maps from the game *Dragon Age: Origins*. The synthesized heuristics perform better than two baselines — Manhattan distance and Octile distance — on each map for the pairs of start and goal states used for synthesis, but this performance does not carry over to unseen pairs of start and goal states for those maps.

1 Introduction

Discovering optimal or near-optimal paths through cluttered maps is an important task for video game agents. These pathing decisions are used by non-playable character (NPC) agents to travel between pre-determined points on a map without having to manually specify an exact route, and by playable character agents to travel from their current position to a position indicated by the player. The speed and optimality of the algorithms used for path discovery can greatly affect the quality of a game. There are many other fields to which optimal pathfinding is of high importance, such as GPS, robotics, logistics, and crowd simulation [1], but this paper will focus on the application to game playing.

A popular method for pathfinding is the A* algorithm [2], which is an extension of Dijkstra's algorithm [3] to include a heuristic for guiding search. The chosen heuristic can have great effect on the performance of A*. One of the most popular heuristics is Manhattan distance, which works very well on open maps, but slightly worse on maps with obstacles such as walls. The performance of a particular heuristic is dependent upon the map it is applied to. A method for automatically generating good heuristics for specific maps would boost the performance of A* on these maps, rather than using a general heuristic such as Manhattan distance. Program synthesis is one method that would work well for automatically generating heuristics.

Program synthesis is the science of automatically generating computer programs [4]. It is a difficult task due to the huge search space of most programming languages. Domain specific languages (DSL) can be created for specific problems to reduce the size of the search space. The DSL contains all operations the designer wishes to include in their domain. How these operations can be combined is outlined by a context-free grammar (CFG). There is a trade-off between the

expressiveness of the DSL, and how large the resultant search space is. Many algorithms can be used for program synthesis, such as guided bottom-up search (BUS).

BUS is a complete search algorithm which builds larger solutions from combinations of smaller solutions. Within the realm of program synthesis, some number of initial programs — usually constants or variables — are defined by the CFG. How to build larger programs is also defined by a separate part of the CFG, and these larger programs then fall under the CFG and are used, along with earlier programs, to continue building programs. This can be continued forever, but usually some time- or size-based bound is used as stopping criteria. Often, certain areas of the BUS search space are much more promising than others. The search can be biased towards these promising areas using methods such as Guided BUS (GBUS) [5].

Guided BUS is a suitable algorithm for synthesizing A* heuristic functions for specific video game maps. The improved heuristics synthesized in this way will allow video game agents to navigate their environments with increased efficiency. This paper first provides a background on the A* and GBUS algorithms. Next, the program synthesis details are provided; this encompasses the choices of DSL and CFG, along with reasoning for both choices. Details of the specific implementations used for experiments are provided. Finally, the empirical experiments performed are explained, and the results of these experiments are presented and analyzed.

2 Background

2.1 The A* Algorithm

The A* algorithm [2] is an extension of Dijkstra's algorithm [3] with a heuristic guiding search. Both algorithms are best-first search algorithms for finding the paths between nodes in a graph. The A* algorithm assigns a cost to each node in the graph. Starting from a specific node, each node connected to the current node by an edge is revealed and its cost is calculated. The next node to visit is chosen as the node with the smallest cost of all revealed and unvisited nodes. Ties are broken using some metric, such as most recently visited. This is repeated until the *goal* node is visited.

The cost, f, of a node, n, is calculated as follows:

$$f(n) = g(n) + h(n)$$

Of the two function on the right side of this equation, g is usually the cost to travel to the node n, and h is the heuristic used to guide search. The heuristic used will depend on the type of graph being searched. The graph for this application represents a grid-based map. For this scenario, the most popular heuristic is the Manhattan distance [6], which is usually used when the agent is constrained to moving in the cardinal directions. The Manhattan distance is calculated as follows.

$$dx = abs(x(n) - goal_x)$$

$$dy = abs(y(n) - goal_y)$$

$$h_{manhattan}(n) = dx + dy$$

In this equation, *abs* is the absolute value, x and y are the horizontal and vertical positions of node n on the map, and $goal_x$ and $goal_y$ are the horizontal and vertical positions of the goal — where the agent wants to travel.

Manhattan distance can be extended to consider diagonal movements. This is done using the Octile heuristic [7], which is as follows.

$$h_{octile}(n) = max(dx, dy) + D * min(dx, dy)$$

For perfect adherence to diagonal distance, $D = \sqrt{2} - 1$ would be used. Instead, $D = 0.5$ is usually preferred for computational reasons.

2.2 Guided Bottom-Up Search

Program synthesis tasks usually require a DSL and CFG. Possible programs are represented by an abstract syntax tree (AST), where branches on the tree are non-terminal functions from the CFG and leaves are terminal functions. Non-terminal functions are those which require terminal functions to be provided to them for a program to be complete and runnable, while terminal functions are already complete and runnable.

BUS begins with the terminal functions, then continuously expands existing ASTs by applying non-terminal functions to them [8]. Each new AST is compared against all existing ASTs to determine if it is equivalent and if so it is discarded. There are multiple types of equivalence. Two programs are weakly equivalent if they produce the same outputs for some subset of all possible inputs. Two programs are strongly equivalent if they produce the same outputs for all inputs.

Some improvements were suggested to this algorithm by Barke et al. [5]. The first of these improvements was to constrain new programs generated during an iteration of BUS to a specific size. This will ensure the solution found is the smallest possible solution, which is important since smaller programs are usually more computationally efficient and human-interpretable. The size of a program is the number of CFG functions it contains.

The second improvement was to bias the search using a probabilistic CFG (PCFG). The paper was searching for string manipulation programs using a few input-output pairs. Each iteration of search was constrained to a maximum cost, like the size constraint mentioned previously. The cost of each function was calculated based on its probability p as follows.

$$cost = -log_2(p)$$

Initially the probability of each function within the PCFG was equal. Upon finding a program that solved a subset of the input-output pairs, the probabilities of each function was updated based on how many of the subset solution programs it appeared it. This biased the search towards what should be good areas of the search space.

3 Implementation

3.1 The A* Algorithm

The specifics of the implementation of A* used in subsequent experiments are outlined here. The g function was set to equal 0 for all states. It was assumed that a heuristic that performs well on $g = 0$ will also perform well on true values of g used to compute shortest paths. The heuristic function, h, is the program being synthesized by GBUS.

The heuristic function values are stored in an array, rather than the graph format usually used by A* implementations. A graph allows easier computation of the shortest path once the goal has

been found, but since we only care about the number of nodes expanded to find the goal, we can use an array and take advantage of its much faster parallel computation. The heuristic costs for each state on the full map are pre-computed in an array. This array is then normalized to contain values between 0 and 1 inclusive. This is done as follows.

$$costs \leftarrow \frac{costs - min(costs)}{max(costs) - min(costs)}$$

A check is performed before computing this to ensure there will be no division by zero. This normalization helps with checking for duplicates during GBUS, but its main use is so the values of walls and states already visited by the agent can be set higher than 1 so the agent will know not to choose them. The search procedure is as follows (values in brackets were used for experiments).

1. Set $expandedNodes$ to 0. Choose a pair of *start* and *goal* states.
2. Set states in the heuristic costs array, $costsArray$, corresponding to walls on the map to $wallValue$ greater than one (9 was used).
3. Create an $explorationArray$ of equal size to $costsArray$ and filled with $wallValue$.
4. Set $currentState$ to *start*, then repeat the following until $currentState$ is *goal*:
   (a) Set the value of $currentState$ in $explorationArray$ to $visitedValue$ between one and $wallValue$ (7 was used).
   (b) Set states in $explorationArray$ which are adjacent to $currentState$ to their corresponding values in $costsArray$.
   (c) Set $currentState$ as the index of the minimum of $explorationArray$, breaking ties randomly.
   (d) Increase $expandedNodes$ by 1.

Since this procedure will eventually find a solution for some start and goal pairs on maps where it should be impossible, each start and goal pair is checked for validity using Manhattan distance prior to running GBUS.

3.2 Domain Specific Language

To decide on which functions to include in the DSL, the Manhattan and Octile heuristics were analyzed. All functions appearing in these heuristics were used in the DSL. The constant value of 2 was included as the reciprocal of the constant value of 0.5 which appears in the Octile heuristic.

3.2.1 Context-Free Grammar

The CFG is as follows.

$$S \rightarrow goal_x \mid goal_y \mid x \mid y \mid 0.5 \mid 2 \mid E$$

$$E \rightarrow S + S \mid S - S \mid S * S \mid max(S, S) \mid min(S, S)$$

$$A \rightarrow abs(E)$$

3.3 Guided Bottom-Up Search

Search was performed in a similar manner to Barke et al. [5]. Programs were automatically generated as an AST according to the CFG detailed previously. Programs generated within each iteration are constrained to a maximum cost; this will be expanded upon later.

Absolute value programs (A in the CFG) are generated at one cost lower than their actual cost. This was done because some programs which could later become useful if they were within an absolute value were instead pruned from the search during equivalence checking. Without this, for example, $(y - goal_y)$ and $(goal_y - y)$ were being pruned as equivalent to $(goal_x - y)$, so $abs(y - goal_y)$ and $abs(goal_y - y)$ were never generated. Empirically, the increased computation from the additional programs due to this is more than offset by the normalized equivalence checking.

Equivalence checking was performed by taking a 5x5 cutout of the normalized heuristic costs array for each start and goal pair. This 5x5 cutout was taken centered around each start state; if this would extend beyond the edges of the map, the cutout was shifted appropriately. Each of these cutouts were stacked, and the full stack was flattened into a tuple. This tuple was then checked against a set of all unique tuples seen so far. A heuristic was considered weakly equivalent to some previous heuristic if its tuple already existed within the set. These heuristics were discarded.

The cost of new programs was calculated based on an approximation of their performance, along with a regularization component to encourage smaller programs. Since some heuristics are much worse than others when evaluated with A* search, a lot of computation could be saved by truncating evaluation when the total number of states expanded over all state and goal pairs exceeds the best so far. This loses a direct comparison of performance between heuristics, but this can be approximated based on how many state and goal pairs the evaluator saw before termination. This value, i, is used to compute performance cost as follows.

$$cost_{performance} = floor(log_\alpha(i/numPairs + \epsilon)) \tag{1}$$

Where $floor$ is the integer floor function, i is the number of start and goal pairs seen, $numPairs$ is the total number of state and goal pairs, ϵ is some small offset (10^{-4} was used), and the logarithm base α is a hyperparameter.

To encourage smaller programs, a regularization cost was added. This cost is as follows.

$$cost_{regularization} = floor((programSize/\beta)^\gamma) \tag{2}$$

Where $floor$ is the integer floor function, and β and γ are hyperparameters.

Total cost is one plus the sum of the performance and regularization costs. If this cost exceeds the true program size, cost is set to the true program size. The calculation is as follows.

$$cost = min(cost_{performance} + cost_{regularization} + 1, programSize)$$

Where min is the minimum function.

When $i = numPairs$ and $programSize < \beta$, the cost will be zero. This may not make sense but seems to work better empirically.

Since new programs may be too small to use at the search's current cost level, the search should be reset occasionally to make use of these new programs. It was chosen to reset search when the number of states expanded by a new best program is at least 5% less than the number of states expanded by the best program upon the previous reset (or infinity if there have been no resets). All programs in the program list are kept upon reset. A set is kept of all programs which have already been evaluated so programs are not evaluated more than once.

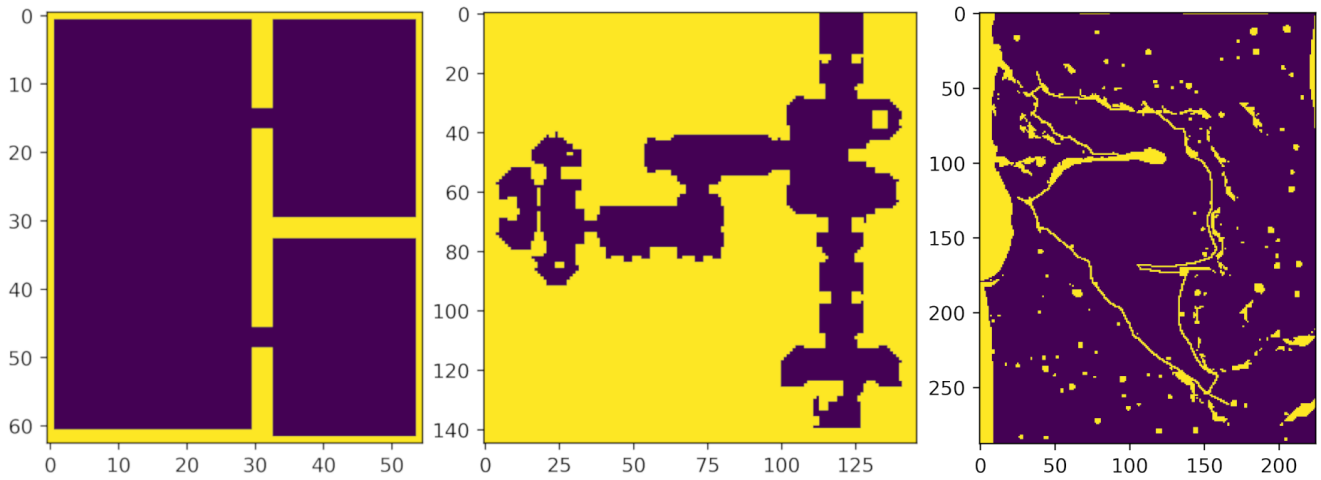

Figure 1: Dragon Age: Origins maps. From left to right — "isound1", "orz302d", "brc501d". Purple indicates a traversable state, yellow an untraversable state.

4 Experiments

4.1 Game Maps

The maps used for experiments are from *Dragon Age: Origins*, a role-playing game developed by BioWare and published by Electronic Arts [9]. Three maps were chosen to run experiments on: "isound1", "orz302d", and "brc501d". These three maps are shown in Figure 1.

4.2 Methodology

GBUS with the A* algorithm described in section 3 was used to synthesize heuristics. For each map, ten start and goal pairs were randomly chosen from all traversable states. Heuristic performance was evaluated using these ten pairs. For hyperparameters, $\alpha = 0.25$ for Equation 1, and $\beta = 10$ and $\gamma = 1.5$ for Equation 2. The cost bound was set at 4 for "isound1", and 3 for "orz302d" and "brc501d". Cost bound decreased with increasing map size since larger maps take longer to evaluate, and therefore search will be slower. Experiments are done once for each map. To test the generality of synthesized heuristics, the best synthesized heuristic for each map is tested against the Manhattan and Octile heuristics on 1000 unseen pairs of start and goal states for that map. This is to increase confidence that the heuristic is actually good on the map, and not overfit to the start and goal pairs used during synthesis.

All code was written in Python 3.8 using NumPy for matrix operations, and experiments were run on an Intel i5-3570k CPU with 8GB of RAM.

5 Results

5.1 Map "isound1"

For this map, the best heuristic found was:

$$max(abs(x - y), abs(x - goal_x)) * (abs(y - goal_y) + abs(abs(x - goal_x) - abs(y - goal_y)))$$

Table 1: Start and goal pairs for map "isound1"

Pair	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Start	(4,14)	(16,24)	(54,1)	(2,48)	(3,26)	(49,49)	(14,48)	(16,27)	(20,16)	(37,33)
Goal	(58,42)	(23,20)	(26,6)	(36,10)	(11,12)	(44,48)	(49,43)	(14,32)	(57,53)	(39,17)

The average number of states expanded by this heuristic was 42.1, compared to 85.8 for the Manhattan heuristic and 70.4 for the Octile heuristic. A total of 449339 heuristics were generated, of which 217004 were evaluated. This took around 1 hour and 30 minutes. The start and goal pairs used for synthesis are shown in Table 1. Numpy array indices are in (row, column) format, so the start and goal indices shown in tables are (y, x) co-ordinates.

The test on 1000 pairs of start and goal states resulted in an average states expanded for Manhattan of 94.16, Octile of 85.48, and the newly synthesized heuristic of 106.23. The new heuristic performed worse than both baseline heuristics on unseen pairs of start and goal states, showing it is likely overfit to the 10 pairs used for synthesis.

5.2 Map "orz302d"

For this map, the best heuristic found was:

$$(abs(x - goal_x) + abs(y - goal_y)) * max(x + x, y * (goal_x - x))$$

The average number of states expanded by this heuristic was 173.4, compared to 278.8 for the Manhattan heuristic and 277.0 for the Octile heuristic. A total of 409359 heuristics were generated, of which 139891 were evaluated. This took around 6 hours and 50 minutes. The start and goal pairs used for synthesis are shown in Table 2.

The test on 1000 pairs of start and goal states resulted in an average states expanded for Manhattan of 165.59, Octile of 139.48, and the newly synthesized heuristic of 157.88. The new heuristic performed worse than the Octile heuristic and slightly better than the Manhattan heuristic on unseen pairs of start and goal states. It is likely overfit to the 10 pairs used for synthesis since it performed much better than the Octile heuristic on those pairs.

5.3 Map "brc501d"

For this map, the best heuristic found was:

$$max(goal_x - x, abs(y - goal_y)) + max(goal_x - x, abs(goal_x - x - abs(y - goal_y)))$$

The average number of states expanded by this heuristic was 742.5, compared to 1342.2 for the Manhattan heuristic and 1358.7 for the Octile heuristic. A total of 21835 heuristics were generated, of which 13134 were evaluated. This took around 10 hours. The start and goal pairs used for synthesis are shown in Table 3.

The test on 1000 pairs of start and goal states resulted in an average states expanded for Manhattan of 1018.45, Octile of 920.86, and the newly synthesized heuristic of 1116.22. The new heuristic performed worse than both baseline heuristics on unseen pairs of start and goal states, showing it is likely overfit to the 10 pairs used for synthesis.

Table 2: Start and goal pairs for map "orz302d"

Pair	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Start	(118,102)	(84,113)	(60,114)	(78,72)	(79,15)	(32,102)	(100,118)	(69,31)	(44,68)	(59,115)
Goal	(20,120)	(70,23)	(106,120)	(115,109)	(121,124)	(50,117)	(70,73)	(123,107)	(71,34)	(74,24)

Table 3: Start and goal pairs for map "brc501d"

Pair	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Start	(170,41)	(16,200)	(105,207)	(89,13)	(271,150)	(150,57)	(107,206)	(209,170)	(75,195)	(63,40)
Goal	(118,223)	(93,128)	(239,71)	(160,212)	(254,100)	(36,174)	(12,212)	(209,207)	(278,106)	(6,143)

6 Conclusion

A program synthesis algorithm, GBUS, was used to synthesize A* heuristics on three maps from the game *Dragon Age: Origins*. A vectorized implementation of A* was used since it allowed much faster evaluation of heuristic performance. The DSL was built based on the functions and constants used in the Manhattan and Octile heuristics. GBUS was used for synthesizing heuristics; program costs were based on an approximation of how well the heuristic performed along with a regularization component to encourage smaller programs. Experiments were run on the three maps, during which heuristics were synthesized using 10 pairs of start and goal states. Synthesized heuristics performed better than baseline heuristics on all three maps for the 10 pairs of start and goal states used during synthesis, but did not perform as well on 1000 unseen pairs of start and goal states.

Some future paths to improve performance are to change the evaluation of heuristics to be more efficient, expand the DSL, optimize the hyperparameters (or use different cost functions altogether), run synthesis on more pairs of start and goal states to reduce the chance of overfitting, and to run synthesis for longer and on better hardware — potentially with parallelization in the code.

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015, 2015.

[2] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[3] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[4] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[5] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[6] Eugene F Krause. *Taxicab geometry: An adventure in non-Euclidean geometry.* Courier Corporation, 1986.

[7] Yngvi Björnsson and Kári Halldórsson. Improved heuristics for optimal path-finding on game maps. *AIIDE*, 6:9–14, 2006.

[8] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International conference on computer aided verification*, pages 934–950. Springer, 2013.

[9] Wikipedia contributors. Dragon age: Origins — Wikipedia, the free encyclopedia, 2021. URL `https://en.wikipedia.org/wiki/Dragon_Age:_Origins`. [Online; accessed 28-March-2021].