Additive Heuristic for Four-Connected Gridworlds

Kenneth Anderson
March Networks
303 Terry Fox Drive
Ottawa, Ontario, Canada

Abstract

Memory-based heuristic techniques have been used to effectively reduce search times in implicit graphs. Recently, these techniques have been applied to improving search times in explicit graphs. This paper presents a new memory-based, additive heuristic that can be used on a type of explicit graph: the four-connected gridworld. The heuristic reduces the number of expanded nodes by up to five times, reduces execution time by up to 29 times, and can efficiently accommodate graph changes.

Introduction

The single-agent search community continually strives to make search techniques more efficient. One effective approach is to improve the heuristics used to guide the search. The improvement of heuristics has received attention because of its general applicability, relative independence to the search technique, and dramatic performance gains. Also, as long as certain conditions are met, A* (or other similar search algorithms) can guarantee finding an optimal (minimal cost) path from a start node to a goal node, if one exists (Hart, Nilsson, and Raphael 1968).

On implicit graphs such as Rubik's Cube, the 15 sliding-tile puzzle, and the K-Pancake puzzle, memory-based heuristic lookup tables are often used to improve IDA* search. Abstraction is used to reduce the graph to a more manageable size; then the abstract graph is solved and the result is stored in memory (Culberson and Schaeffer 1998). Combining multiple memory-based heuristics by adding them together is particularly effective for a variety of puzzle domains (Korf and Felner 2002; Yang et al. 2008).

When searching explicit graphs, such as GPS navigation, pathplanning in games, and robotic motion planning, the A* search algorithm is used quite frequently. Unlike the puzzle domains, however, optimal search in explicit graphs using memory-based heuristics has only recently started to show promising results (Sturtevant et al. 2009).

This paper will present a new abstraction-based, admissible heuristic technique for four-connected gridworlds. The main contributions are threefold. (1) The new heuristic technique is described. (2) Various methods of calculating this heuristic on-demand are compared. (3) This technique is proposed for dynamically changing graphs.

Using the additive heuristic technique in A* search is four times faster than using the Manhattan Distance heuristic on video game graphs and 29 times faster on room graphs. The memory requirements are modest and graph updates can be accommodated quickly.

The remainder of the paper discusses relevant previous work, describes the new additive heuristic, experimentally compares on-demand heuristic calculation methods, and proposes areas open for future development.

Background

Pattern databases (PDBs) dramatically improve search performance on the 15-sliding tile puzzle by using a combination of backward search and graph abstraction to improve heuristic values (Culberson and Schaeffer 1998). By abstracting the full graph representing the 15 sliding-tile puzzle to a smaller graph, they are able to completely solve the smaller graph and store the costs to the goal in a database. Using these costs as a heuristic in the forward search reduces the number of node expansions by three orders of magnitude.

Additive pattern databases further improve upon PDBs by leveraging multiple abstractions to build multiple PDBs (Korf and Felner 2002; Yang et al. 2008). Furthermore, the databases are cleverly designed to enable the heuristics to be added together admissibly. Additive PDBs provide an impressive performance boost and are currently state-of-the-art on many puzzle testbeds.

The main drawback of PDBs is the computation required to calculate the PDBs. On puzzle domains, this cost can often be amortized over all the forward searches because the goal node is fixed. However, in most explicit graph searches, such as those in computer games and GPS route planning, the goal node changes for every search instance. In such cases, one of the following two techniques can be used: true distance heuristics or on-demand heuristic calculation.

True distance heuristics (Sturtevant et al. 2009) precompute a database of costs in the original graph. Then they combine database queries with simple arithmetic to produce an admissible heuristic between any two nodes in the graph. When used for pathfinding in an eight-connected gridworld, they achieve a 29-fold reduction in node expansions. The

precomputation of the database of costs, however, involves multiple searches in the original graph. As a result, the time required to build this database is even more significant than for PDBs. Thus, there is an implicit assumption that the original graph does not change.

Another technique for finding optimal solutions for any start and goal node is to only calculate the heuristics as needed. We refer to this as on-demand heuristic calculation. An *instance-dependent pattern database* (IDPDB) can be used to calculate a heuristic on-demand (Felner and Adler 2005). For each search instance, an IDPDB is calculated by using a backward search in the abstract space. This database stores heuristic values that are used in the forward search.

Hierarchical search also calculates heuristics on-demand by using multiple, hierarchical graph abstractions (Holte et al. 1996). For each heuristic requested, a forward search in the abstract space calculates the cost. This abstract search may, in turn, request heuristic values from another abstraction level, prompting further forward searches. Both techniques have proven effective in puzzle domains.

If there is a desire to accommodate changing graphs, such as with GPS route planning or the fog-of-war in computer games, then rebuilding the databases from the ground up must be avoided. Although explicit, dynamic graphs are not examined, IDPDBs and hierarchical search should be able to efficiently accommodate graph changes.

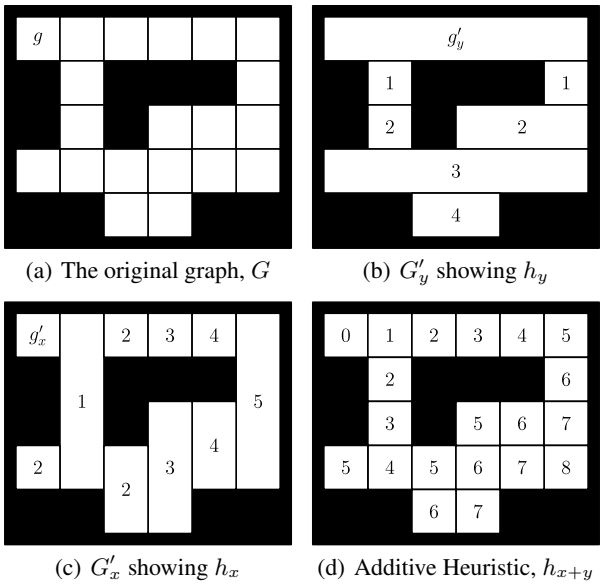
Additionally, there are abstraction-based techniques that can handle graph changes (Sturtevant and Buro 2005; Botea, Müller, and Schaeffer 2004). With these techniques, a hierarchy of abstractions is used to quickly find suboptimal paths. A change in the original graph is efficiently dealt with by propagating changes to nearby neighbors and up the hierarchy. Unfortunately, these techniques cannot guarantee finding optimal paths.

The following section introduces a novel, additive, memory-based, heuristic technique for finding optimal paths on a four-connected gridworld. The heuristic technique can efficiently deal with graph changes.

Additive Heuristic for Gridworld

The domain under consideration is a four-connected gridworld. Each location on the 2-D grid is either 'unoccupied' or 'occupied' (by a wall). This world is represented as a graph, G, where each unoccupied grid location is a node. Each node can be connected to up to four other neighboring nodes (North, South, East, or West) by an undirected edge of weight one. A *path* from a start node, t, to a goal node, g, consists of a series of edges connecting the two nodes. The path's *cost* is the sum of the weights of the edges on the path. Our objective is to find a minimal-cost path from t to g (see Figure 1(a)).

A* search is a popular search technique used in explicit graphs. A* expands nodes in priority, based on the sum of the cost from the start node and a *heuristic* (an estimate of the cost to the goal). As long as the heuristic is *admissible* (does not overestimate the cost to the goal), A* is guaranteed to find an optimal path (Hart, Nilsson, and Raphael 1968). Furthermore, if the heuristic is *consistent* (does not change more than the weight of an edge between two nodes), it will expand the minimum number of nodes possible for that heuristic (Dechter and Pearl 1985).



(a) The original graph, G (b) G'_y showing h_y

(c) G'_x showing h_x (d) Additive Heuristic, h_{x+y}

Figure 1: The original graph (a) shows the goal node. The abstract graphs (b) and (c) show the abstract goal nodes and heuristic values. The final graph (d) shows the result of adding the heuristics together.

The Manhattan Distance heuristic, h_{MD}, is a simple, commonly used heuristic in gridworlds. It is admissible and consistent, making it a good heuristic for A* search. The Manhattan Distance from any node n to the goal node g can be computed by taking the sum of the positional differences in the x and y directions. For example, if the position of t is $(0, 2)$ and the position of g is $(5, 1)$, then $h_{MD}(t, g) = |0 - 5| + |2 - 1| = 5 + 1 = 6$.

The new heuristic presented in the following is similar to h_{MD} in that there are two heuristics, h_x and h_y, that are added together admissibly.

X-Heuristic

To create h_x, first construct an abstract graph, G'_x by taking into account only the x direction.

- Select any node in the graph G and merge this node with its neighbor nodes in the y direction (North and South), if they exist, to create an abstract node. Repeat until no further merges are possible. All duplicate edges between any two abstract nodes are merged to create an abstract edge whose weight is equal to the lowest weighted edge (in this case the weight is one).
- Create a mapping from each node $n \in G$ to its corresponding abstract node $n'_x \in G'_x$, including g to g'_x. Similarly, map each edge $e \in G$ to its corresponding abstract edge $e'_x \in G'_x$.
- Find the cost of each abstract node in G'_x to the abstract goal using a breadth-first search.

For any node n in G, there is a mapping to an abstract node n'_x in G'_x, where the cost from n'_x to g'_x is an underestimate of the cost from n to g (see Figure 1(c)). This cost is called $h_x(n, g)$.

Y-Heuristic

h_y is created similarly; graph G'_y is constructed by paying attention to only the y direction. The costs in G'_y are used as the heuristic h_y (see Figure 1(b)).

Additive Heuristic

The new heuristic h_{x+y} is created by adding together the costs in the abstract graphs (Figure 1(d)):

$$h_{x+y}(n, g) = d(n'_x, g'_x) + d(n'_y, g'_y)$$

where d is the cost in the corresponding abstract graph from abstract node n' to the abstract goal node g'. Also, $n \in G$ maps to $n'_x \in G'_x$ and $n'_y \in G'_y$. Similarly, $g \in G$ maps to $g'_x \in G'_x$ and $g'_y \in G'_y$.

The h_{x+y} is admissible because the set of edges that maps from G to G'_x does not overlap with the set of edges that maps from G to G'_y. Therefore, the edges are not counted more than once in the abstract graph.

The h_{x+y} is also consistent. The heuristic h_x is equal to the cost to g'_x in the abstract graph G'_x. Fundamentally, solution costs in the abstract graph are consistent: for any two abstract nodes connected by an abstract edge, the change in the solution cost to the abstract goal is not more than the weight of the connecting abstract edge. The same is true with h_y and G'_y. Any edge in G maps to one and only one edge in either G'_x or G'_y. Therefore, the heuristic h_{x+y} between any two connected nodes cannot change more than the weight of an abstract edge. Because the weight of any edge in G, G'_x, or G'_y is one, the heuristic is consistent.

Finally, h_{x+y} is at least as informative as h_{MD}. h_{MD} can actually be viewed as a simplified case of h_{x+y}, where all nodes with the same y-coordinate map to the same abstract node in G'_x, and all nodes with the same x-coordinate map to the same abstract node in G'_y.

On-Demand Heuristic Calculation

The additive heuristic is generated by performing a backward search in each abstract space. There are multiple methodologies that can be used for generating the heuristic. The first technique is to completely solve the full space using either breadth-first or depth-first search, and store the costs to the abstract goal node. This technique is used for PDBs (Culberson and Schaeffer 1998; Korf and Felner 2002). The database covers the entire abstract graph (Figure 2).

However, careful consideration will reveal that calculating heuristic values over the entire abstract graph is not necessary. Only the heuristic values for the nodes that actually occur in the forward search need to be calculated. This observation is important because the time spent calculating the heuristic is being considered as part of the search time. An alternative is to use *partial pattern databases* (PPDBs), whereby the backward search is started and paused when it reaches the abstract node that requested the heuristic value (Anderson, Holte, and Schaeffer 2007). This implementation is slightly different than originally described because the backward search resumes after receiving a request for an unexpanded node (Figure 2).

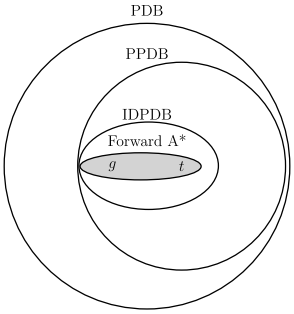


Figure 2: Coverage of on-demand heuristics

IDPDBs, also called Reverse-Resumable A*, take this approach further by using an available heuristic to focus the search effort (Felner and Adler 2005; Silver 2005). IDPDBs use a heuristic while searching backward from the abstract goal to an abstract start. The key is to resume the search by expanding nodes with the *same heuristic* (the start and goal remain the same). The search continues to progress until the requested abstract node is closed, yielding a minimal-cost to the abstract goal. If the search in the original graph mimics the backward search in the abstract graph, then this process should be efficient (Figure 2).

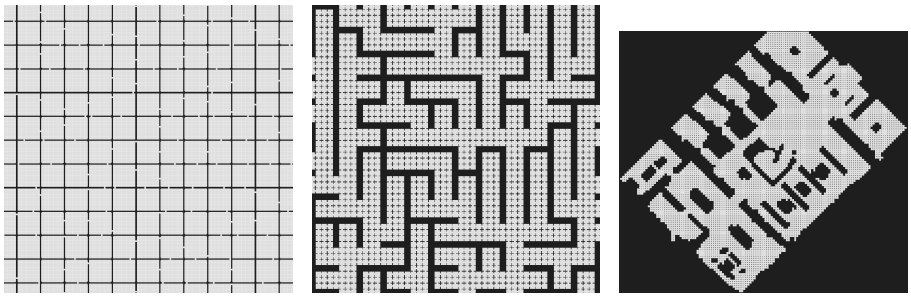
IDPDBs are able to leverage heuristics, if they exist in the abstract graph. For the graph abstractions G'_x and G'_y, admissible, consistent heuristics do indeed exist. The heuristic for G'_x is the x component of the Manhattan Distance heuristic. The heuristic for G'_y is likewise the y component of the Manhattan Distance heuristic.

It should be noted that using PDBs, PPDBs, or IDPDBs results in exactly the same heuristic values; it is merely the technique used to calculate the heuristic that varies. Additionally, in puzzle domains it is typical to stop the backward search when memory is full. However, because the explicit graph is already held in memory, enough memory is assumed to be available to perform a search of the full abstract space.

Results

For these tests, three types of graphs are compared: room graphs, maze graphs, and video game graphs (Figure 3). The room graphs (a) consist of 256 connected rooms on a 512x512 gridworld. The maze graphs (b) consist of corridors of width three, on a 512x512 gridworld. The video game graphs (c) are from Baldur's Gate (BioWare Corp 1998), and range in size from 52x54 to 320x320. All graphs are four-connected.

Two main heuristics are compared in these tests: the Manhattan Distance heuristic (h_{MD}), which is considered the baseline, and the additive heuristic (h_{x+y}). The additive heuristic, however, can be calculated using any of



(a) Room (b) Maze (c) Video game

Figure 3: Example graph types tested.

the following techniques: h_{x+y}^{PDB}, which uses full, breadth-first backward searches; h_{x+y}^{PPDB}, which uses on-demand breadth-first backward searches; h_{x+y}^{IDPDB}, which uses on-demand A* backward searches.

Table 1 shows the averaged results of A* search using different heuristics on different graph types. The graph types are shown in column one. Column two shows the heuristic used for each test. For each row, one hundred graphs are tested with one hundred search instances (randomly selected start and goal nodes), totaling 10,000 search instances each.

Abstract Graph and Heuristic Properties

Column three in Table 1 shows the average abstract graph size of a single abstract graph in relation to the original graph size. This size gives us an idea of how much effort would be required to search the abstract graph in order to construct the additive heuristic; smaller abstract graph sizes usually indicate less search effort to calculate the heuristic. Keep in mind, however, that there are two abstract graphs (G'_x and G'_y).

The Manhattan Distance heurisitic is simple to calculate and does not require an abstract graph at all, so its abstract graph size is zero. The abstract graph size of the additive heuristic tends to be smaller when the graph has large open (unwalled) areas. Open areas cause more nodes to map to each abstract node, leading to smaller abstract graphs, as is the case with video and room maps. However, the maze graphs do not have any open areas and suffer from large abstract graph sizes.

Column four in Table 1 shows the average heuristic value over the entire graph for each search instance. Higher heuristic values usually lead to fewer node expansions during search. As previously mentioned, the heuristic values of the additive heuristic are always higher than Manhattan Distance. However, for the room and maze graphs the average heuristic value is almost two times the value of the Manhattan Distance heuristic.

The following section will examine the number of nodes expanded. Based on this initial analysis, additive heuristic should be most effective on the room graphs because of the small abstract graph sizes and the high heuristic values.

Nodes Expanded

Column five in Table 1 shows the average number of nodes expanded during the search instances; this number includes nodes expanded in the forward search on the original graph as well as the nodes expanded in the backward searches on the abstract graphs.

As expected, using the additive heuristic expands fewer nodes on average than using the Manhattan Distance heuristic on room graphs. Most notably, using IDPDBs expands five times fewer nodes than search using the Manhattan Distance heuristic. This reduction is caused by the small abstract graph sizes and the high heuristic values.

On maze and video game graphs, using the full PDBs actually expands more nodes than using the Manhattan Distance heuristic. This increase is primarily because each abstract graph is searched entirely to calculate the heuristics. The heuristic calculation dominates the search effort and results in poor performance. When using PPDBs, the average number of nodes expanded improves to about the same as using Manhattan Distance. Using IDPDBs gets the fewest average number of nodes expanded, about half as many as search using the Manhattan Distance heuristic.

Figure 4 examines the number of nodes expanded as a function of solution cost. The search instances are grouped together according to their solution cost (buckets of size 50). The number of nodes expanded is averaged within each group. Only the room graph results are shown, as all the graph types had similar results.

The number of nodes expanded using the Manhattan Distance heuristic increases steeply as the solution cost increases. This behavior is expected, as longer solution costs are generally more difficult.

The number of nodes expanded when using PDBs, however, increases very slowly. When compared to using the Manhattan Distance heuristic, using PDBs expands many more nodes on the small search instances, but expands many fewer nodes on the larger instances. This behavior is caused by the heuristic calculation being independent of the solution cost; the full PDBs are calculated even if the solution cost is one! Once the heuristic is calculated, the number of nodes expanded by the forward search (shown by h_{x+y}) is very small. Therefore, the search effort is dominated by calculating the heuristic.

Graph type	Heuristic type	Avg abstract graph size	Avg heuristic value	Avg nodes expanded	Avg time (msec)
Room	h_{MD}	0.00%	363.23	31731.17	870.39
	h^{PDB}_{x+y}	13.28%	730.94	31282.49	133.56
	h^{PPDB}_{x+y}	13.28%	730.94	11301.38	53.42
	h^{IDPDB}_{x+y}	13.28%	730.94	5576.38	30.40
Maze	h_{MD}	0.00%	428.19	27754.58	888.24
	h^{PDB}_{x+y}	33.33%	736.92	66141.69	306.22
	h^{PPDB}_{x+y}	33.33%	736.92	28515.35	136.96
	h^{IDPDB}_{x+y}	33.33%	736.92	12866.90	67.69
Video game	h_{MD}	0.00%	44.90	389.21	4.43
	h^{PDB}_{x+y}	14.75%	55.12	578.27	2.30
	h^{PPDB}_{x+y}	14.75%	55.12	309.90	1.42
	h^{IDPDB}_{x+y}	14.75%	55.12	203.98	1.11

Table 1: Abstract graph size and heuristic value averaged over 100 maps. Number of nodes expanded and execution time averaged over 10,000 A* search instances.

When using PPDBs, the number of nodes expanded during heuristic calculation is reduced. Most notably, for small search instances few nodes are expanded, which is an improvement over using PDBs. In large instances, the number of nodes expanded using PPDBs approaches, but should never exceed, the number of nodes expanded using PDBs.

The behavior of search using IDPDBs is similar to that of the PPDBs, but even better in terms of the number of nodes expanded. This improvement is using a heuristic in the backward searches, causing fewer nodes to be expanded.

Timing

Column six in Table 1 shows the average execution time in milliseconds. The tests were performed on a single-core, AMD Athlon XP 2500+ (1.83 GHz CPU) with 1.00 GB of RAM. The implementation was in C++ using the standard template library.

The timing results show that in all three graph types considered, the search using the additive heuristic is faster than the search using the Manhattan Distance heuristic. This result may seem a little unexpected, considering that the number of nodes expanded using the PDBs is sometimes larger than the number of nodes expanded using the Manhattan Distance heuristic. Keep in mind, however, that the number of nodes expanded is the sum of the number of nodes expanded in the forward search and the two backward searches. So when using the Manhattan Distance heuristic, only one large forward search is involved. For the additive heuristic, however, one small forward search and two mid-sized backward searches are involved.

A single large (forward) A* search instance will generally have a larger open list (fringe) than two small (backward) search instances. In addition, the abstract graphs have a smaller branching factor (average number of edges) than the original graph because half of the possible edges are eliminated. This combination of factors causes the forward search in the original space using the Manhattan Distance heuristic to have a substantially larger open list than the backward searches used to calculate the additive heuristic.

In these tests, the implementation of the open list relies on the standard template library's priority queue. The access time for adding and removing nodes from the open list is therefore dependent on the size of the open list: $O(log(m))$, where m is the size of the open list. Since the open list for the forward search using the Manhattan Distance heuristic can get quite large, it actually slows down the search algorithm.

The resulting performance difference is most noticeable in the larger graphs: the average search time using IDPDBs on maze graphs is 7.6 times faster than when using Manhattan Distance, and on room graphs the average search time is almost 30 times faster! On the smaller video game graphs, the average search time is only 4 times faster.

Complexity of Graph Updates

Adding or removing an edge in graph G is trivial. However, updating the abstract graphs involves remapping nodes from G to their corresponding abstract nodes in G'_x and G'_y. In a four-connected gridworld, an edge must be vertical or horizontal. The abstract graphs only retain edges in either the horizontal or vertical directions. Therefore, the number of remappings caused by adding or removing an edge is limited to only one of the abstract graphs.

When adding a new edge, the number of node remappings is limited to the number of nodes mapped to the same abstract node. The number of node remappings is inversely proportional to the relative size of the abstract graphs (column three in Table 1). For room graphs the average number of node remappings is 7.53 nodes; for maze graphs it is 3.00 nodes; for video game graphs it is 6.78 nodes. The number

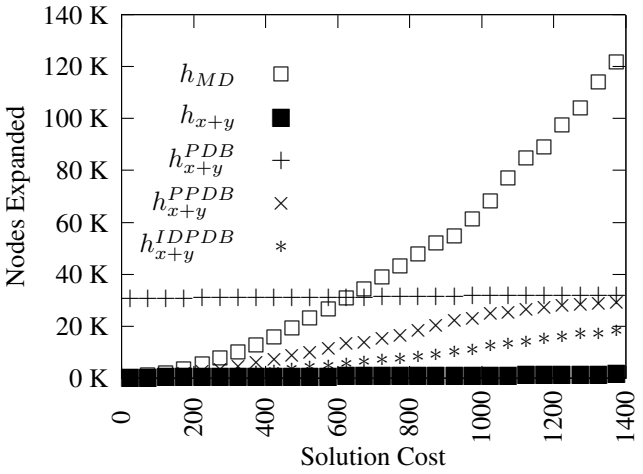


Figure 4: Average number of nodes expanded vs. solution cost on room graphs.

of edges that potentially require remapping is limited to two edges per remapped node.

When removing an edge, a new abstract node is potentially created. To accommodate the new abstract node, the average number of node remappings and edge remappings is twice the number of remappings required to add a new edge.

For the tested graph types, the number of updates is quite reasonable and h_{x+y} can be updated very quickly when adding or removing edges in graph G. It should be noted, however, that in the worst case (a graph of a straight line) all nodes might have to be remapped. So the characteristics of the graph do need to be taken into consideration.

Conclusions and Future Work

In this paper, a new additive heuristic was presented that can be used to find optimal paths on a four-connected gridworld. The additive heuristic is very strong; it dramatically reduces the number of nodes expanded in the forward search. However, searches in two abstract graphs are required to calculate the heuristic.

The heuristic can be calculated efficiently using on-demand techniques. Of the techniques tested, using IDPDBs is the most effective, reducing the total number of nodes expanded by a factor of five on room graphs and a factor of two on video game graphs. On maze graphs the heuristic cannot be calculated as efficiently and the total number of nodes expanded is reduced by only a factor of 1.9.

Because of the implementation details of A*, the execution times are even more noteworthy than the number of nodes expanded. Search on room graphs using the additive heuristic has a 29-fold reduction in execution time as compared to using the Manhattan Distance heuristic, and search on maze graphs has a 7.6-fold reduction. Search on video game graphs reduced the execution time by a factor of 2.5.

The heuristic is so effective in these tests that the search performance is dominated by the calculation of the heuristic itself. One future course of action is to try to further reduce the search effort in the abstract graph. As a result of the abstraction technique, the structure of the abstract graphs is quite different than the structure of the original graph. Specifically, the abstract graphs are sparsely connected. This property can be leveraged by using existing algorithms to further simplify the abstract graphs and speed up the search (Demyen and Buro 2006).

This type of abstraction technique lends itself quite readily to 2-D graphs in a four-connected gridworld. It has yet to be seen how well this technique works on eight-connected worlds where diagonal edges provide an extra complication. In addition, expanding this technique to larger dimensional spaces would make it more relevant to other domains such as robot motion planning, time-based search, and 3-D search.

Acknowledgments

I would like to express my appreciation to the reviewers, to Robert Holte for his comments, and to Nathan Sturtevant and BioWare for providing graphs.

References

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In *SARA*, 20–34.

BioWare Corp. 1998. Baldur's gate. Interplay.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1:7–28.

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

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery* 32(3):505–536.

Demyen, D., and Buro, M. 2006. Efficient triangulation-based pathfinding. In *AAAI*, 942–947.

Felner, A., and Adler, A. 2005. Solving the 24 puzzle with instance dependent pattern databases. In *SARA*, 248–260.

Hart, P. E.; Nilsson, N. J.; 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.

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI*, 530–535.

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

Silver, D. 2005. Cooperative pathfinding. In *AIIDE*, 117–122.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *JAIR* 32(1):631–662.