The Closed List Is an Obstacle Too

Ariel Felner[1], **Shahaf S. Shperberg**[2], **Hadar Buzhish**[1]
[1] SISE Department, Ben-Gurion University, Be'er-Sheva, Israel
[2] CS Department, Ben-Gurion University, Be'er-Sheva, Israel
felner@bgu.ac.il, shperbsh@post.bgu.ac.il, hadarbuzi@gmail.com

Abstract

The baseline approach for optimal path finding in 4-connected grids is A* with Manhattan Distance. In this paper we introduce an enhancement to A* (called BOXA*) on grids which does not need any preprocessing and only needs negligible additional memory. The main idea is to treat the closed-list as a dynamic obstacle. We maintain rectangles which surround CLOSED nodes and calculate an admissible heuristic using the fact that an optimal path from a given node must go around these rectangles. We experimentally show the benefits of this approach on a variety of grid domains.

Introduction

Optimal path finding in grids is important in AI and robotics. In this paper we limit the discussion to 4-connected 2D grids. The baseline approach is to use A* with Manhattan Distance (MD). Nevertheless, a large number of enhancements were suggested over the years, most of which require a preprocessing phase and/or additional memory to store smart lookup tables which speed up the search (Sturtevant et al. 2015). At one extreme, some methods calculate and store the *all pairs shortest paths* information (Botea and Harabor 2013). At the other extreme, some methods such as *jump point search* (Harabor and Grastien 2011) do not require any preprocessing or additional memory.

This paper introduces an enhancement to baseline A* on grids which does not need any preprocessing and only needs negligible additional memory. The main idea is to treat closed nodes as obstacles. Dynamic lists of rectangles which surround the closed list are maintained. We then calculate an admissible heuristic using the fact that an optimal path from any given node in OPEN must at least go around these rectangles. Finally, we provide experimental results that demonstrate the benefits of this approach. This paper is a proof-of-concept for using the closed-list to improve the heuristic of open nodes. We believe that this idea can be also applied to other gird settings (e.g. 8-connected and 3D), and maybe even to other domains.

Motivation and Definitions

We assume that there exists a consistent *base heuristic* (h_{base}). Throughout this paper we use MD as h_{base}. The consistency of h_{base} means that a node is always expanded and closed with its best g-value and that it will never be reopened (Zhang et al. 2009). A motivating example appears in Figure 1(left); blue nodes are in CLOSED and green nodes are in OPEN (similarly, in all our figures). Assume A* is executed from s (the start state) to g (the goal state). Further assume that node c was already closed with $g(c)$, that node x has just been generated and that we are seeking for $h(x)$. We now have the following lemma:

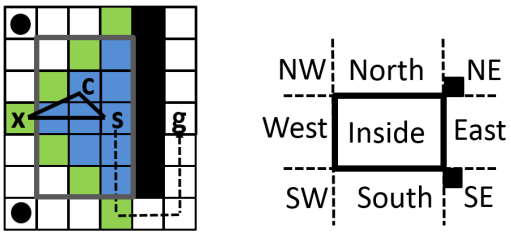

Figure 1: Motivating example

Lemma 1. *If x is on an optimal path P from s to g which was discovered by A*, then c will never appear on the remaining part of P after x.*

Proof (by contradiction): Assume that $P = \{s \rightsquigarrow x \rightsquigarrow c \rightsquigarrow g\}$ is an optimal path from s to g. But since c was already closed and h_{base} is consistent P cannot be discovered by A* as A* never reopen nodes when the heuristic is consistent. In fact, in this case c was closed with its best g-value $g(c) \leq g(x) + d(x, c)$. So, instead of P, A* will find an alternative path $P' = s \rightsquigarrow c \rightsquigarrow g$ via the original expansion of c with $c(P') \leq c(P)$ (because $g(c) \leq g(x) + d(x, c)$).[1] □

We note that the minimal requirement on a heuristic h assuring that A* finds an optimal solution is that there exists one optimal path P where $\forall_{x \in P}(h(x) \leq h^*(x))$. That is, h is required to be admissible only on P (Karpas and Domshlak 2012). As a result, nodes that are not on this unique path P may have overestimating h-values. In particular, consider a node $y \notin P$ when it is generated. Ideally, if an oracle predicts that $y \notin P$ it would be safe to set $h(y) = \infty$ and never

[1] If the smallest edge cost is $\epsilon > 0$ then $c(P') < c(P)$ and $g(c) < g(x) + d(x, c)$.

expand y. In this setting, A* will only expand the nodes on the optimal path and reach the goal directly.

Therefore, when A* generates a node x the question we actually ask is: *if x is on an optimal path, what is a lower bound on the remaining optimal path to the goal?* We call such lower bound *path admissible*. Consequently, when a node x is generated, we want a lower bound on a path that does not pass through any CLOSED node (because an optimal path that includes x cannot include any CLOSED node after x). In this sense, for this particular search from s to g, CLOSED can be treated as an obstacle.

In the remaining of this paper we focus on finding such a lower bound. We denote it by h_{BOX}.

Rectangles Around Closed Nodes

Consider again the example in Figure 1(left) and observe the gray rectangle which is the smallest rectangle that surrounds CLOSED. Simple properties of 4-connected grids imply that any shortest path that does not pass in any CLOSED node (blue nodes) is equivalent in its length to the path that goes around the rectangle. Therefore, h_{BOX} calculates the shortest path to the goal that goes around the rectangle.

Any rectangle r in a grid divides the grid into 9 zones based on their locations relative to r as depicted in Figure 1(right). These zones can be (1:) cardinal to the rectangle (e.g., north, east, south, west) (2:) diagonally to the rectangle (e.g., north-west), or (3:) inside the rectangle. We say that a rectangle *blocks* cell x from cell y (and vise versa) if they are on opposite cardinal zones (i.e., north vs. south or east vs. west). Similarly, we define four *pivot* points which are diagonally adjacent to the corners of the rectangle. The black squares in Figure 1(right) present the north-east and the south-east pivots of the rectangle.

Definition 1 (h_{BOX}(x,r)). *Given that cell x is cardinal to a rectangle r which surrounds closed cells, let $PV(r,x)$ be the two pivots that are on the same cardinal side as x (the black circles in Figure 1(left) are $PV(r,x)$).*

If x and goal are not blocked by r then:

$h_{BOX}(x,r) = h_{base}(x)$

If x and goal are blocked by r (they are in opposite cardinal directions) then:

$h_{BOX}(x,r) = \min_{q \in PV(r,x)}(h_{base}(x,q) + h_{base}(q))$

Lemma 2. *$h_{BOX}(x,r)$ is path admissible.*

Proof: As explained above, if x is on an optimal path to the goal then this path must not enter any of the CLOSED nodes. The length of the minimal path from x to the goal that does not pass through any CLOSED node is therefore a lower bound on the remaining optimal path. This is equivalent to a path that goes around the rectangle that surrounds the CLOSED nodes. The shortest path that passes through one of the pivots, $PV(r,x)$, is a lower bound for this. □

It is easy to see that $h_{BOX}(x,r) \geq h_{base}(x)$. As a result, it is likely that the number of expanded nodes will be smaller with $h_{BOX}(x,r)$ than with $h_{base}(x)$ (but not necessary, see Holte (2010)). Next we introduce our algorithm BOXA* and a number of enhanced variants.

BOXA*

BOXA* maintains a list R_D of disjoint, unconnected rectangles which surround CLOSED nodes for each direction $D \in \{\mathbf{N}orth, \mathbf{E}ast, \mathbf{S}outh, \mathbf{W}est\}$ relative to g. An example is shown in Figure 2(a) where R_N is colored red, R_W is colored purple and R_E is colored orange. R_S is empty. For node x, $h_{BOX}(x, r_D)$ is calculated for every rectangle $r_D \in R_D$ that *blocks* x from g. That is, x and g are in opposite cardinal zones with respect to r_D.

Note that if x is on the same row or the same column as g then there is only one direction D that may have rectangles that block x from g. Otherwise, x is diagonal to g and there are two directions that might have rectangles which block x from g. For example, if x is southwest to g then rectangles from both R_S and R_W may block x from g. Each rectangle produces a lower-bound on the solution cost, thus the maximal h_{BOX} value among them can be used as the (path admissible) heuristic value. Thus, the full (single-parameter) version of h_{BOX} is defined with respect to a given cell x and all rectangles that block x from g:

$$h_{BOX}(x) = \max_{r_d \in R_D \text{ s.t. } r_d \text{ blocks } x \text{ from } g} h_{BOX}(x, r_d)$$

For example, assume that cell x is generated in Figure 1(left). While MD(x) = 5, $h_{BOX}(x) = 11$ as we must make 6 vertical moves to get to one of the pivots and continue to the goal, due to the (single) west rectangle r_W. Finally, note that if a node x is inside a rectangle r when computing $h_{BOX}(x,r)$, only a sub-rectangle of r that does not contain x should be considered. For example, in the left side of Figure 2(a), x_1 is within the (single) west rectangle r_W. Thus, when computing $h_{BOX}(x_1, r_W)$ only the sub-rectangle that contains the three nodes right of x_1 is relevant.

Maintaining the Rectangles

In the beginning of the search, there are no rectangles as CLOSED is empty. Since CLOSED grows whenever a node is expanded, the rectangles can possibly change after each expansion. There are three cases when a node x is expanded which are covered next.

Case 1: If a node x_1, which is inside a rectangle r is expanded, then r remains unchanged. This case is demonstrated in Figure 2 where the west rectangle, r_W is shown before (a) and after (b) the expansion of x_1.

Case 2: If a node x_2, which borders a rectangle r is expanded, then r needs to be extended to also include x_2. This process is shown in Figure 2 where the west rectangle, r_W is shown before (b) and after (c) the expansion of x_2.[2]

Case 3: If a node x_3, which does not border any rectangle $r_D \in R_D$ in a relevant direction D (based on the location of x_3 with respect to g) is expanded, then a new rectangle r'_D is created and added to R_D. This process is shown in Figure 2. The south rectangle, r_S (in yellow) is shown before the expansion of x_3 (c). Then, after x_3 is expanded, a new rectangle r'_S is created (d). In principle, due to this case there

[2]In theory, if a node which borders two rectangles in the same direction is expanded, these rectangles can be merged. However, this never happened in our experiments.

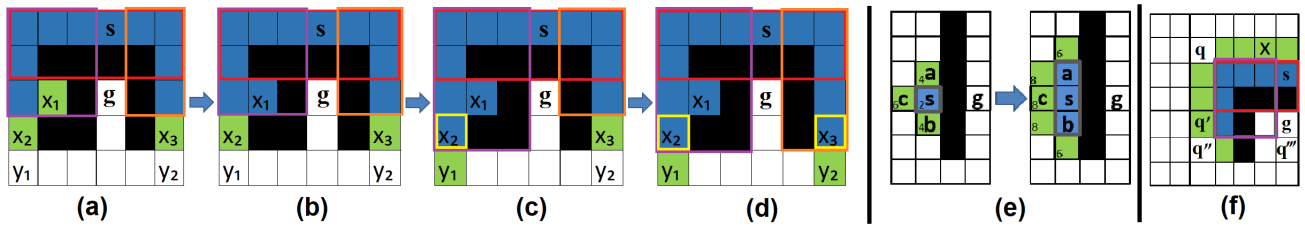

Figure 2: (a:) initial state (b:) no enlarging (c:) enlarging (d:) multiple rectangles (e:) Lazy Evaluation (f:) Recursive h_{BOX}

can be several disjoint rectangles in R_D for some direction D (as just shown for R_S). In our experiments there was usually only a single rectangle in each direction, and never more than two. Moreover, when computing $h_{BOX}(x)$, not every rectangle from the relevant list R_D blocks x from g. For example, in Figure 2(d), even though y_1 and y_2 are south to g, none of the south rectangles blocks them. Finally, note that when s is expanded, then CLOSED grows from empty to include s. Therefore, s does not border any rectangle and the initial rectangles are constructed due to case 3.

Implementation of BOXA*

BOXA* is mainly based on A*, with some additional overhead for computing the new heuristic and maintaining the rectangles. When BOXA* generates a node n, the following operations are performed:

1. Find the zone of n with respect to the goal; this is done by a simple comparison of the (x, y)-coordinates of n with those of g.
2. Iterate over the rectangles in each relevant direction (induced by n's zone), and check if they block n from the goal, i.e., check if n is either adjacent to or inside any of the rectangles. In the latter case (n is inside a rectangle r), consider only a sub-rectangle of r that doesn't contain n.
3. For every rectangle that blocks n from the goal, compute h_{BOX}. The computation of h_{BOX} is composed of computing the distance to the two pivots, and applying h_{base} (MD) to each pivot.
4. Take the maximum between the lower-bounds induced by the different rectangles (step 3).

When a node n is expanded, some of the rectangles in step 2 need to be extended (as explained in Section). Furthermore, if one of the relevant directions induced by the zone of n (step 1) had no rectangles that blocks n from g, a new rectangle is generated. Both these operations (extending a rectangle and creating a new rectangle) are computationally inexpensive. All of the above generation and expansion operations of nodes consume a time that is at most linear in the number of rectangles. In practice, the number of rectangles from each direction was usually 1 (and never more than 2). Thus, the additional overhead of BOXA* compared to A* for every node generation and expansion is effectively constant. The memory consumption of BOXA* is also linear in the number of rectangles, as each rectangle is stored using two coordinates (the northwest and southeast corners). Thus, the overall additional memory consumption of BOXA* is constant in practice.

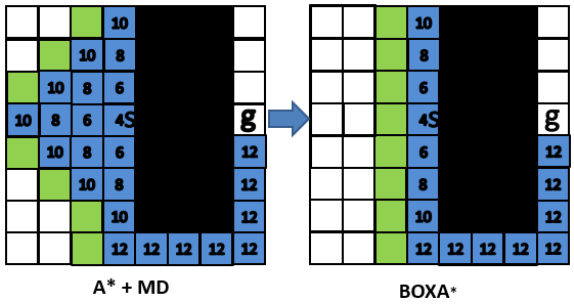

Figure 3: A* vs BOXA* with a 3×7 obstacle

Reduction in Node Expansions of BOXA*

Potentially, BOXA* may significantly reduce the number of nodes expanded, but this greatly depends on the specific instance and on the exact location of obstacles with respect to s and g. Consider the grid in Figure 3 where the start s and the goal g are in opposite sides of a rectangular obstacle of height H and width W. Numbers inside cells are their f-values. A* will expand the entire triangle left to start before it goes around the obstacle as shown in Figure 3(left). This will expand $H^2/2 + H/2 + W + 2$ nodes. In contrast, BOXA* will only expand $3H/2 + W + 2$ nodes as shown in Figure 3(right). When H is large and W is small BOXA* will have a quadratic reduction in the number of nodes expanded compared to A*. For example, consider a rectangular obstacle of size $1,000 \times 4$. In this case BOXA* expands $1,506$ nodes while A* expands $500,506$ nodes. However, when $H = 4$ and $W = 1,000$, BOXA* expands $1,008$ nodes while A* expands $1,014$ nodes. Furthermore, assume that we swap the locations of s and g and search from g to s. Here, A* will expand all of the nodes in the rightmost column and then go below the obstacle directly to s. BOXA* (and any other algorithm with no preprocessing) will not be able to prune any node and will do the exact same work as A*. In general BOXA* will be most efficient if s and g are blocked by a long obstacle, but this is not known a priori to the solver without any preprocessing.

Enhancements of BOXA*

We next cover two enhancements for BOXA*.

Lazy Expansion of Nodes

We note that when x is chosen for expansion, we can optionally re-calculate $f(x)$ based on the new shape of the rectangles in the R_D lists of the relevant directions. If this increases $f(x)$ above the minimal f-value in OPEN then x may remain in OPEN with its new f-value along the same principle used by Lazy A* (Tolpin et al. 2013). Therefore, we call this variant BOXA$^*_l$. This *change-priority()* operation is lighter than a full expansion. In *change-priority()* we re-insert a node x with its new f-value. In a standard heap implementation of a *priority queue* this incurs $O(log(M))$ time where M is the number of nodes in OPEN. By contrast, expansion of a node is removing it from OPEN and adding b children. This incurs a larger overhead than *change-priority()* because it involves $b+1$ (one deletion and b additions) operations on OPEN, each of them takes $O(log(M))$ time. In addition, it involves the generation of b new nodes. Therefore, *change-priority()* is a lighter operation. However, as shown by Tolpin et al. (2013), *change-priority(x)* will only be beneficial if x remains in OPEN and is never expanded. This will happen if the new $f(x)$ is larger than C^*, the cost of the optimal solution. In this case, the expansion of x is saved. If, however, it turns out that x will be expanded in later stages then *change-priority(x)* is redundant and only incurs extra overhead. Thus, activating *change-priority(x)* is helpful only for some of the nodes.

Consider the example in Figure 2(e). The left side shows the grid after expanding node s. The number in each cell is its f-value. There is only one closed rectangle and it contains s only. For node c, $g(c) = 1$ and $h(c) = 5$ because it needs to go around the rectangle to reach g. In the next two steps nodes a and b are expanded and the rectangle is updated to contain {s,a,b} as shown in Figure 2(e) on the right. Now the best node in OPEN will be c. When extracting node c its heuristic is re-evaluated to $h(c) = 7$ and therefore we have that $f(c) = 8$. There are now two options. The first is to expand c right away and generate its children. However, since we have nodes in OPEN with $f(n) = 6$, the second option is to choose not to expand c but place it in open with $f(c) = 8$ via the *change-priority()* function. In fact, in this example, c might never be expanded.

Recursive h_{BOX}

In Definition 1 (of h_{BOX}), the h-value of reaching from a pivot q to the goal is MD (h_{base}). However, instead of using MD, h_{BOX} can be recursively applied on the pivots as well. Thus, we define h_{BOX_r}, a recursive version of h_{BOX}, as follows:

$$h_{BOX_r}(x,r) = \min_{q \in PV_r(x)} (h_{base}(x,q) + h_{base}(q,q') + h_{BOX_r}(q'))$$

where q' is the other corner of r that is adjacent to q in the cordiality opposite of x . For example, if x is north to r and q is the northwest corner of r, then q' is the southwest corner of r. The full (single-parameter) version of h_{BOX_r} is thus defined:

$$h_{BOX_r}(x) = \max_{r_d \in R_D \text{ s.t. } r_d \text{ blocks } x \text{ from } g} h_{BOX_r}(x, r_d)$$

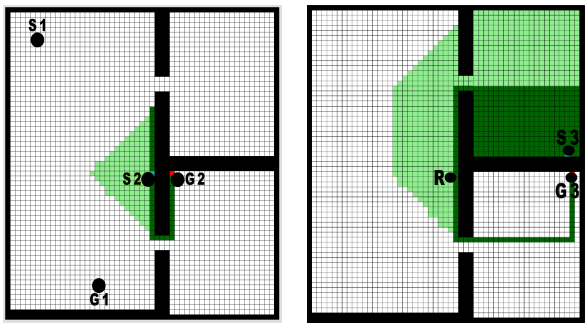

Figure 4: isound1. Left: scenarios 1, 2. Right: scenario 3

Figure 2(f) demonstrates the contribution of h_{BOX_r} when $h(x)$ is computed. The north rectangle blocks x from g. There is only one pivot q as the other corner is outside of the map. Since h_{BOX} applies MD on the pivot nodes, $h_{BOX}(x,r) = h_{base}(x,q) + h_{base}(q,g) = 3+7 = 10$. In contrast, when computing $h_{BOX_r}(x)$, h_{BOX_r} is applied recursively on q', thus $h_{BOX_r}(x) = h_{base}(x,q) + h_{base}(q,q') + h_{BOX_r}(q')$. When computing $h_{BOX_r}(q')$, the west rectangle is considered and q'' becomes the new pivot, thus $h_{BOX_r}(q') = h_{base}(q',q'') + h_{base}(q'',q''') + h_{BOX_r}(q''')$. Finally, $h_{BOX_r}(q''') = h_{base}(q''',g) = 1$, as it is not bounded by any rectangle. Therefore, $h_{BOX_r}(q') = 1+4+1 = 6$. As a result, $h_{BOX_r}(x) = 3+3+6 = 12$. We use BOXA$^*_r$ to denote the recursive variant of BOXA* (that uses h_{BOX_r} instead of h_{BOX}), and BOXA$^*_{lr}$ to denote the variant that is both lazy and recursive.

Experimental Results

We compared all our variants on maps from Dragon Age: Origins (DAO) (all *brc* maps and the *isound1* map) and on Mazes with different corridor widths (1, 2, 4); all are from the *movingai* repository (Sturtevant 2012). The improvement of the BOXA* variants over A* varies dramatically along different maps and scenarios. In some cases, for example, when there is a line-of-sight between the start and the goal, A* cannot be improved. However, in other cases there is almost quadratic improvement in the number of node expansions, as explained above. We demonstrate this trend on three scenarios of the `isound1` map. The start and goal cells of Scenario 1 are labeled S_1 and G_1 in Figure 4 (left). These cells have line-of-sight and thus *all* algorithms proceed directly to the goal. Scenario 2 (S_2 and G_2) is also shown in Figure 4 (left). The light green cells are those that were expanded by A* and the dark green cells are those expanded by both BOXA$^*_{lr}$ and A*. Clearly, a large improvement is observed. A* expanded 214 nodes while BOXA$^*_{lr}$ expanded 40 nodes, an improvement factor of 5.4. Another major improvement is also evident for Scenario 3 (shown in Figure 4 (right)). Here, A* expanded 1,058 nodes while BOXA$^*_{lr}$ expanded 363 nodes, a larger reduction in expansions compared to Scenario 2, but a smaller improvement ratio of 2.9.

Table 1 reports the number of node expansions and gener-

	Type	Best improvement factor				Average node expansions					Average node generations				
		BOXA*	BOXA$^*_l$	BOXA$^*_r$	BOXA$^*_{lr}$	A*	BOXA*	BOXA$^*_l$	BOXA$^*_r$	BOXA$^*_{lr}$	A*	BOXA*	BOXA$^*_l$	BOXA$^*_r$	BOXA$^*_{lr}$
DAO	brc	3.3	3.9	3.3	**4.0**	6,796	5,931	5,808+427	5,902	**5,605**+673	12,655	7,958	7,834	7,970	**7,637**
	isound	3.0	3.2	3.1	**5.4**	235	197	186 + 25	195	**178** + 28	490	330	321	329	**312**
Mazes	w=1	**1.6**	**1.6**	**1.6**	**1.6**	24,673	23,280	23,265 + 23	23,119	**22,996** + 64	24,709	23,317	23,302	23,156	**23,033**
	w=2	2.0	2.1	2.0	**3.2**	27,088	24,767	24,722+103	24,526	**23,808**+174	40,677	26,285	26,217	26,030	**25,250**
	w=4	2.2	2.2	2.2	**2.7**	31,276	29,298	29,244+257	29,120	**28,638**+404	54,804	33,931	33,842	33,701	**33,137**

Table 1: Best improvement factor and average node expansions and generations for the different BOXA* variants

ations for all our variants on all map types. The first columns present the improvement factor over A* of the best scenario on each map type. Basic BOXA* reduced the number of node expansions by a factor that ranges from 1.6 to 3.3 while BOXA$^*_{lr}$ further increased the improvement factor up to 5.4 (as shown in Figure 4(right) for scenario 2). The following columns show the average number of node expansions and node generations over all maps and scenarios. For BOXA$^*_l$ and BOXA$^*_{lr}$ the number of change-priority calls of lazy expansion is shown after the plus sign (+). Since in many scenarios there was a line-of-sight between the start and the goal, the average improvement is relatively modest. BOXA* resulted in an average reduction of 9.3% in node expansions and 29% in node generations. Each of the BOXA* enhancements further improve the results and BOXA$^*_{lr}$ resulted in an average reduction of 13.3% in node expansions and 33.1% in node generation. Importantly, all BOXA* variants were never worse than A* in any map and scenario in terms of node expansions/generations. Thus there is no risk in using BOXA* and its enhancements in terms of node counts.

While the BOXA* variants are never worse in terms of node expansions, this is not the case in terms of runtime. The operations that maintain the rectangles and calculate h_{BOX} are asymptotically constant, but they incur a larger CPU overhead compared to A*. This is especially evident in small maps, where the overhead of node-insertions is small due to the small size of the open-list (heap). Therefore, the runtime of all BOXA* variants was larger than A* on average. Nonetheless, BOXA* can still be better in terms of runtime for larger maps. For example, when multiplying size of the isound1 map by 10, *all* BOXA* variants had a better runtime than A* on average, and in some scenarios the improvement was by up to a factor of 4. Finally, our implementation of BOXA* is relatively basic, but more low-level implementation optimizations are likely to speed up the runtime significantly. Meanwhile, we only recommend using BOXA* and its variants for domains that 1) have many obstacles, thus the start and the goal are not likely to share a line-of-sight; and 2) the maps in the domain are either large or the cost of node generations is high (e.g. if it requires a physical sensing).

Conclusions

We presented an effective method to strengthen MD on 2D grids without preprocessing and without significant additional memory. In terms of node expansions and generations, all BOXA* variants are never worse than A* and in many scenarios they might provide a significant improvement even in time. Improvements vary by instance based on the shapes of the obstacles and the relative locations of start and goal.

We believe that this work is mainly a proof of concept for a much larger research. First, BOXA* can be combined with other orthogonal approaches, especially those that do not need preprocessing nor significant additional memory such as *jump point search* (Harabor and Grastien 2011). Then, BOXA* can be generalized to 3D grids where boxes will replace the rectangles and to 8-connected grids where polygons will replace the rectangles. Finally, we believe that the idea of using the closed-list to improve the heuristic for generated nodes can be applied to other polynomial domains (e.g. roadmaps) and even to exponential domains (e.g. puzzles). For example, one way to generalize to other domains (e.g., exponential domains) is to embed them (e.g., by fastmap (Cohen et al. 2018)) into a Euclidean, even 2D, domain and then use our heuristics there.

Acknowledgments

This work was supported by Israel Science Foundation (ISF) grant #844/17 to Ariel Felner and Eyal Shimony, by BSF grant #2017692, by NSF grant #1815660 and by the Frankel center for CS at BGU.

References

Botea, A.; and Harabor, D. 2013. Path Planning with Compressed All-Pairs Shortest Paths Data. In *ICAPS*, 293–297.

Cohen, L.; Uras, T.; Jahangiri, S.; Arunasalam, A.; Koenig, S.; and Kumar, T. K. S. 2018. The FastMap Algorithm for Shortest Path Computations. In *IJCAI*, 1427–1433.

Harabor, D. D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding On Grid Maps. In *AAAI*, 1114 – 1119.

Holte, R. C. 2010. Common Misconceptions Concerning Heuristic Search. In *SoCS*, 46 – 51.

Karpas, E.; and Domshlak, C. 2012. Optimal Search with Inadmissible Heuristics. In *ICAPS*, 92–100.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Trans. Comput. Intell. AI Games* 4(2): 144–148.

Sturtevant, N. R.; Traish, J. M.; Tulip, J. R.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The Grid-Based Path Planning Competition: 2014 Entries and Results. In *SoCS*, 241–251.

Tolpin, D.; Beja, T.; Shimony, S. E.; Felner, A.; and Karpas, E. 2013. Toward Rational Deployment of Multiple Heuristics in A. In *IJCAI*, 674–680.

Zhang, Z.; Sturtevant, N. R.; Holte, R. C.; Schaeffer, J.; and Felner, A. 2009. A* Search with Inconsistent Heuristics. In *IJCAI*, 634–639.