Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning

Peter Yap and **Neil Burch** and **Rob Holte** and **Jonathan Schaeffer**
Computing Science Department
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
blueberrypete@gmail.com, {nburch, rholte, jonathan}@ualberta.ca

Abstract

We present three new ideas for grid-based path-planning algorithms that improve the search speed and quality of the paths found. First, we introduce a new type of database, the Local Distance Database (LDDB), that contains distances between boundary points of a local neighborhood. Second, an LDDB-based algorithm is introduced, called Block A*, that calculates the optimal path between start and goal locations given the local distances stored in the LDDB. Third, our experimental results for any-angle path planning in a wide variety of test domains, including real game maps, show that Block A* is faster than both A* and the previously best grid-based any-angle search algorithm, Theta*.

1 Introduction

Path planning on maps is an important problem in many domains, particularly in robotics, GPS navigation and commercial computer games. Fast path planning can be challenging for a number of reasons. First, some domains have to deal with a dynamically changing map. For example, military simulations and real-time strategy (RTS) games operate in a highly dynamic map world, where paths (bridges, doors) or obstacles (buildings, vehicles) can be suddenly moved, built or destroyed. Second, the domain may require paths to be computed for multiple agents. For example, a RTS game (like *Starcraft*) must handle the pathfinding requests of up to eight competing human players, each capable of controlling an army of up to 200 units: a worst case total of 1600 pathfinding calls which must be handled in real-time over a network. These paths generally must be solved (or partially solved) in milliseconds. For these (and other) reasons, the pathfinding algorithm must be as fast as possible.

Grids are standard for highly dynamic multi-agent domains due to their speed. Grids are also common in single-agent robotics (high resolution grids result in better quality paths). A limitation of grids is that searches are limited to moving from a cell to one of its eight closest neighboring cells (an octile graph). This confines the path to 45° or 90° turns, which may result in loss of path optimality and an unaesthetic (or unhuman-like) route. Algorithms like Field D* (Ferguson and Stentz 2006) and Theta* (Daniel et al. 2010) attempt to overcome this by considering paths with heading changes of any-angle. Theta* produces the best quality paths, but is much slower than A*.

In this paper, we introduce a new algorithm, Block A*, that uses a new type of pre-computed database, a Local Distance Database or LDDB, to achieve high-performance pathfinding. Block A* achieves its performance by dealing with blocks of grid cells ($m \times n$ regions of cells) at a time, in contrast to traditional pathfinding algorithms which manipulate individual cells (1×1 units). Information about all possible distances across a block is pre-computed and stored in the LDDB. Different local distance measures used in this pre-computation lead to different types of searches, including those that produce paths constrained to 4 directions (tile), 8 directions (octile), or any-angle.

This research makes the following contributions:

1. We introduce a new type of database, the Local Distance Database (LDDB) whose entries are the actual distances between boundary points of a local neighborhood.
2. A new A*-based pathfinding algorithm, Block A*, is presented that manipulates blocks of grid cells, rather than a single cell at a time.
3. Block A* is data-driven in the sense that it uses the LDDB to compute g-values. By changing the LDDB, different types of searches can be performed (e.g., Manhattan, octile, or any-angle paths).
4. Experimental results show that Block A* using an any-angle LDDB is superior to A* and Theta*.

2 Local Distance Database (LDDB)

A Local Distance Database, or LDDB, stores the exact distance between the boundary points of a local region of the search space. The search space is grouped into regions of $m \times n$ contiguous grid cells. During a search, the LDDB is queried to find the change in g-values that the search will see when it enters a block at one location on the boundary of the block and then exits from all of the block's other boundary locations. Assume that x and y are cells on the boundary of a block. Querying LDDB$[x, y]$ will return the cost of the least-cost path between these two cells. For every possible block configuration, the LDDB stores the lowest path cost for each (x, y) pair.

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In a grid-based pathfinding problem, the search space is usually stored as a two-dimensional array, where each cell can be accessed by its Cartesian coordinates. In the simplest (and most common) case, each cell has a binary value: obstructed (value 1) or unobstructed (value 0). For a $b \times b$ block of cells, there are $2^{b \cdot b}$ possible patterns of grid obstructions. This exponential growth in size means that b will have to be quite small, but a 4×4 block LDDB is already very effective, and even a simple 2×2 block LDDB outperforms A* by a speed factor of 2. In practice, only a small subset of these patterns are actually used in a particular search, but storing all patterns allows us to handle any search domain, including dynamically changing ones, without having to build a new LDDB.

Symmetry can be used to reduce the number of entries in the LDDB (e.g., for some domains rotational symmetry can reduce the size by four). For most domains there will be states in the LDDB that are unreachable and can be eliminated. While the Block A* algorithm (described in the next section) can handle LDDBs that do not contain all patterns, for simplicity a full LDDB is used in this work. The focus of this paper is on using LDDBs to improve search performance; no effort has been made to minimize the LDDB size.

The LDDB is constructed such that for each possible pattern of grid obstacles in a block, an optimal path cost will be stored for every boundary cell of the block to every other boundary cell on all four sides of the block. For simplicity, consider a LDDB in a grid where we can only move in four directions and each move has unit cost. The optimal paths for each pattern in the LDDB can be found by breadth-first search; for a given pattern the computation is inexpensive because the local region is tiny (2×2 to 5×5 in this work). For the LDDB in this search space, with blocks of size $b \times b$, there are at most $4(b-1)$ ingress cells and $4(b-1)-1$ egress cells. Thus, the LDDB has a total of $2^{b \cdot b} \times 4(b-1) \times (4(b-1)-1)$ entries. With only rotational symmetry compression, a 4×4 LDDB fits under 3MB.

Thus far, we have not discussed the values in the LDDB. They could be calculated assuming that grid movements are constrained to horizontal and vertical moves. However, this is not a constraint of the LDDB; it is dictated by the application domain. The LDDB computation can be done using octile (8-way) movements, if that is permitted by the application. Similarly an LDDB can be constructed to find paths with realistic turn radiuses, as often required in robotics applications. The values stored in the LDDB do not change the search algorithm (Block A*).

3 Block A*

Block A* is A* adapted to manipulate a block of cells instead of a single cell at a time. Each entry on its OPEN list is a block that has been reached but not yet expanded, or which needs to be re-expanded because new or cheaper paths to it have been found. The priority of a block on the OPEN list is called its *heap value*. Like A*, the basic cycle in Block A* is to remove the OPEN entry with the lowest heap value and expand it. The LDDB is used during expansion to compute g-values for the boundary cells in the block being expanded.

Algorithm 1 Expand $curBlock$. Y is the set of $curBlock$'s ingress cells.

```
1:  PROC: Expand(curBlock, Y)
2:  for side of curBlock with neighbor nextBlock do
3:     for valid egress node x on current side do
4:        x' = egress neighbor of x on current side
5:        x.g = min_{y∈Y} (y.g + LDDB(y, x), x.g)
6:        x'.g = min(x'.g, x.g + cost(x, x'))
7:     end for
8:     newheapvalue = min_{updated x'} (x'.g + x'.h)
9:     if newheapvalue < nextBlock.heapvalue then
10:       nextBlock.heapvalue = newheapvalue
11:       if nextBlock not in OPEN then
12:          insert nextBlock into OPEN
13:       else
14:          UpdateOPEN(nextBlock)
15:       end if
16:    end if
17: end for
```

Pseudocode for Block A* is given in two parts. Algorithm 1 is the pseudocode for how Block A* expands a block. This is the biggest difference between A* and Block A* and will be discussed first. The main procedure for Block A*, Algorithm 2, will then be discussed.

The operation of Algorithm 1 will be explained via the example in Figure 1. In this example there are only unit cost horizontal and vertical moves. In the centre of the left side of the figure is $curBlock$, the 5×5 block with corners $A2$, $A6$, $E6$ and $E2$, that is about to be expanded. The black tiles are obstacles. $curBlock$ is on the left edge of the search space so has only three neighbor blocks; the grey cells are the cells in the neighbor blocks that are immediately adjacent to cells in $curBlock$ (e.g., cells $A1$ to $E1$ are the topmost cells of the neighboring block under $curBlock$).

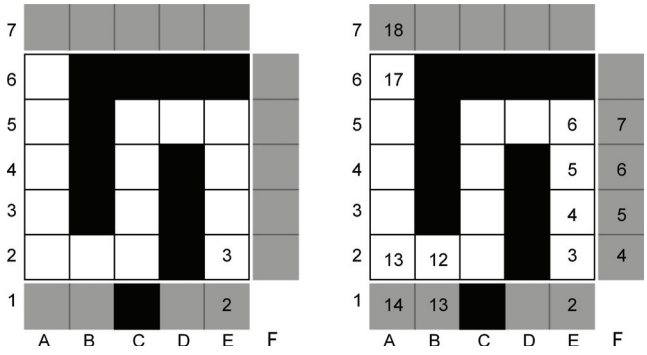

Figure 1: Expanding a block: before (left) and after (right)

$curBlock$'s ingress cells (the set Y) are the boundary cells of $curBlock$ that have a different g-value now than they had the previous time $curBlock$ was expanded (if this is the first time $curBlock$ is being expanded then boundary cells with a finite g-value are ingress cells). In this example, the only ingress cell is $E2$ with a g-value

of 3. The first step in expanding $curBlock$ is to identify its valid egress cells (line 3), which are the unobstructed boundary cells that are adjacent to unobstructed cells in a neighbouring block (which are called their egress neighbors (line 4)). For example, cell $B2$ is a $valid$ egress cell but $C2$ and $D2$ are not. Then the LDDB entry for $curBlock$'s pattern of unobstructed and obstructed cells is retrieved and g-values for all $valid$ egress cells ($A2, A6, B2, E2, E3, E4$, and $E5$) are calculated based on the g-values of the ingress cells and the distance between the ingress and egress cells given by the LDDB (line 5). For example, $A6.g = min_{x \in ingress}(x.g + LDDB(x, A6), A6.g)$ = $E2.g$+14 = 3+14=17. From $A6$'s g-value of 17, a g-value for $A6$'s egress neighbor $A7$ is computed ($A7.newg$ = $A6.g$+1=18). The actual g-value of an egress neighbour is only changed if the new g-value is an improvement over the existing one (line 6). For example, $E1$'s existing g-value of 2 is not improved by the g-value of 3 + 1 = 4 calculated from the egress cell $E2$. The egress neighbour cells whose g-value has changed here will be the ingress cells for their block ($nextBlock$) if it is expanded.

The right side of Figure 1 shows the g-values after expanding $curBlock$. To finish the expansion, we compute the heap value for each neighbouring block and, if necessary, insert it into OPEN (line 12) or update its priority on OPEN (line 14). For simplicity, assume that h = 0 for all tiles. The block to the right of $curBlock$ will be added to OPEN with a heap value of $min(4, 5, 6, 7) = 4$, the block on top will be added with value 18, and the block below will be added *again* with a value of $min(13, 14) = 13$. The reason for the latter value is that $E1$'s previous g-value of 2 has not improved, and a g-value is not used in the heap value calculations if it was not updated (line 8). Even if a tile's g-value is not used in the heap value calculations, it is stored for use later when and if its block is expanded. If the block being put in OPEN is already there we only update OPEN if its heap value has improved (line 9).

Algorithm 2 Block A*

```
 1: PROC: Block A* (LDDB, start, goal)
 2: startBlock = init(start)
 3: goalBlock = init(goal)
 4: length = ∞
 5: insert startBlock into OPEN
 6: while (OPEN ≠ empty) and
    ((OPEN.top).heapvalue < length) ) do
 7:    curBlock = OPEN.pop
 8:    Y = set of all curBlock's ingress nodes
 9:    if curBlock == goalBlock then
10:       length = min_{y∈Y}( y.g+dist(y, goal), length)
11:    end if
12:    Expand( curBlock, Y )
13: end while
14: if length ≠ ∞ then
15:    Reconstruct solution path
16: else
17:    return Failure
18: end if
```

Algorithm 2 is the main procedure for Block A*; it is similar to A* except for the special handling of the initial processing of the blocks containing the goal cell ($goalBlock$) and the start cell ($startBlock$). The call to $init(start)$ (line 2) computes the shortest path from the start cell to every reachable boundary cell in $startBlock$. Line 3 does the same for the goal cell and $goalBlock$. Special care is taken if $start$ and $goal$ are in the same block.

Figure 2 illustrates how Block A* works. The search space is the 15×10 grid shown in #0. S and G are the start and goal cells. Blocks are 5×5, using the Manhattan heuristic. Cells that A* would visit have been shaded. In this contrived example, A* has to visit almost all the cells.

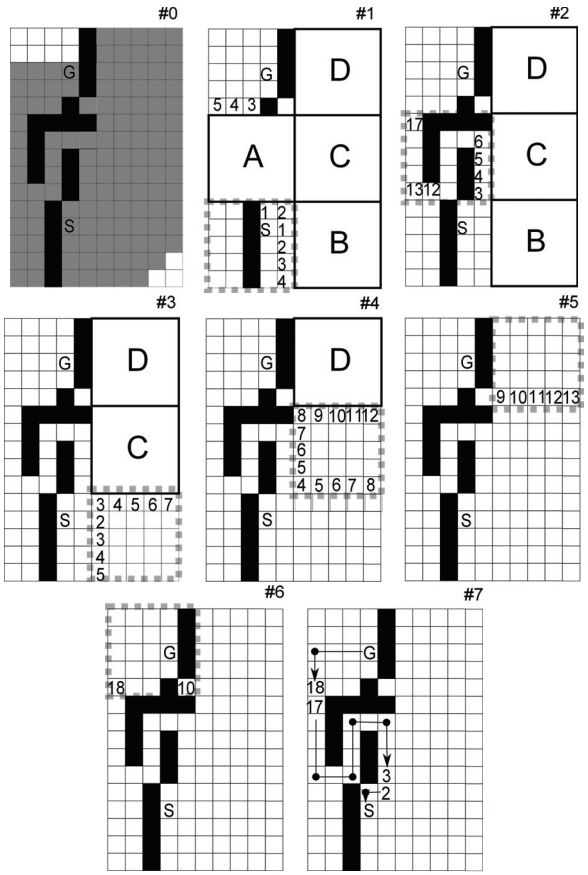

Figure 2: Block A* example

#1 shows the results of the initialization process just described. $startBlock$, with the g-values shown, is added to OPEN. The main loop of Algorithm 2 then begins by removing $startBlock$ from OPEN and expanding it as described above. There is only one valid egress cell from $startBlock$ to block A, the top-right cell in $startBlock$ with a g-value of 2. Hence, block A is added to OPEN with a heapvalue of $2 + 1 + 8 = 11$ (2+1 is the g-value of the egress neighbour in A, and 8 is its h-value). There are five valid egress cells from $startBlock$ to block B so B is added to OPEN with a heap value of $min(2 + 1 + 10, 1 + 1 + 11, 2 + 1 + 12, 3 +$

$1+13, 4+1+14) = 13$. At this point the LDDB entries for A and B have not yet been queried.

A has the smallest heap value and becomes the next $curBlock$. This expansion of A was described in detail in Figure 1, and the right side of that figure shows the g-values that are computed for cells in A and the neighbouring blocks ($goalBlock$ is above A, B is to the right of A, and $startBlock$ is below A). By contrast, in #2 we see the g-values computed for A but not for the neighbouring blocks. This is what will be shown in every part of Figure 2. A's neighbors are now added to OPEN. There is only one egress cell from A to $goalBlock$, with a g-value of 17, so $goalBlock$ is added to OPEN with a heap value of $17+1+5 = 23$ (5 is the exact distance from this egress cell's neighbour in $goalBlock$ to the goal cell, which is known from the initialization step for $goalBlock$). From A one can also go to C; C is added to OPEN with the heap value $min(6+1+6, 5+1+7, 4+1+8, 3+1+9) = 13$. There are three valid ways to go from A back to $startBlock$. Returning through $startBlock$'s top-right cell does not improve that cell's g-value, so it is ignored. The other two paths back to $startBlock$ reach parts of the block that are not reachable from the start cell by paths wholly within $startBlock$. Since one of these might conceivably be on the shortest path to the goal, $startBlock$ is added back onto OPEN with a heap value of $min(12+1+10, 13+1+11) = 23$.

B and C both have the smallest heap value (13) on OPEN at this point. Suppose B is chosen for expansion (#3). B provides new ways to enter C, and therefore updates some of the g-values in C, but the heap value calculated from these g-values, $min(3+1+9, 4+1+10, 5+1+11, 6+1+12, 7+1+13) = 13$, is the same as its current heap value.

From B we can also return to $startBlock$ but doing so does not decrease any g-values in $startBlock$ or add any new ones. C is expanded next; it has ingress nodes reached from both A and B (#4). D is added to OPEN with a heap value of $8+1+4 = 13$. A and B are not re-inserted into OPEN because none of their cell's g-values have improved.

Next, D is expanded (#5) and $goalBlock$ is reached for a second time, with an improved heap value of $9+1+3 = 13$. We know, from the initialization step, that there is no path to the goal cell entirely within $goalBlock$ from the cell in $goalBlock$ that has just been reached from D, but that does not preclude it from being on the optimal path to goal, so it must not be ignored.

$goalBlock$ is now at the head of the OPEN list and is expanded (#6). It has two ingress cells. No path to the goal is found from D, but a path from A is found with a value $17+1+5 = 23$. Like A*, Block A* cannot terminate immediately; it must continue until it is certain that no cheaper path exists. At the head of OPEN is $startBlock$ (reached from A) with a heap value of 23, therefore every remaining path must be at least length 23. Thus the path we found must be optimal. Block A* now terminates and reconstructs the path using back pointers (#7). The path from 17 to 3 in block A is stored as a set of inflection points (circles) in the LDDB, and these back pointers were updated during the search. The path from G to 18 of $goalBlock$ and 2 to S in $startBlock$ were found by $init()$.

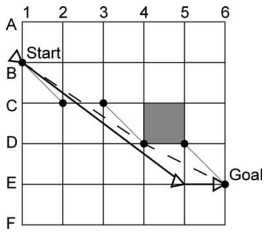

Figure 3: Any-angle search results on a vertex block

4 Any-Angle Search

Thus far we have searched only on the cells of the grid. A slight modification is needed to be able to perform any-angle searches which operate on the vertices of the grid (but the obstacles are still cell-based). In Figure 3, the obstacle is the grey box bounded by vertices $C4, C5, D4$, and $D5$. This is a cell, but the search is on vertices. To convert Block A* to search on vertices, one need only construct a LDDB that takes exterior vertices of a block as input, rather than using the exterior cells. For vertex blocks, the exterior vertices of a block are shared with another block[1] and the corner vertices are shared with 3 other blocks. In contrast, the exterior cells of a cell-based block are not shared. In other words, Block A* does not change; only the database used.

In Figure 3, the shortest path from $B1$ to $E6$ using cells is 8. With an any-angle LDDB using vertices, Block A* finds the optimal path $B1-D4-E6$ (dashed line) with a cost of 5.84 (it is retrieved from the 5×5 LDDB). Theta* will find the suboptimal path $B1-E5-E6$ after some computation (open arrows; cost is 6). A* will find a zig-zag-like path $B1-C2-C3-D4-D5-E6$ (filled circles; cost is 6.24). A* is constrained to heading changes that are $45°$, resulting in many directions changes (twice as many, in this example).

To aid in recomputing the path once found by Block A*, the inflection points used in the optimal paths can be saved in the LDDB. For example, in Figure 3 the inflection point $D4$ is stored for the (ingress, egress) pair $(B1, E6)$. The $E6$ entry will point to $D4$, and $D4$ will point to $B1$ (assuming they are all on the optimal path).

A* will always find the optimal path, relative to the grid that it searches on. Similarly, Block A* will find the optimal path relative to the LDDB that it uses. However, in an any-angle sense, these "optimal paths" are not necessarily any-angle optimal. Block A* uses optimal local (block) information to help improve its path quality, but it lacks the global information needed for an optimal answer.

Theta* is essentially A* except that the parent of a vertex in Theta* is not confined to its neighbor. Theta* finds paths of any-angle while searching on a grid that is bounded by $45°$ angles. Normally for A*, when the node s' is expanded its child s'' has a pointer back to its parent s'. However in Theta*, for each expansion of vertex s', it calls a line of sight (LOS) check for every child s'' of s'. If a LOS check from the child s'' to s, parent(s'), is successful, then this implies that a straight line can be drawn from s'' to s and the parent

[1] In Algorithm 1, line 6, cost(x,x') = 0 for vertices.

Data Set	Algorithm	Distance	Expanded	Time (s)
Random 0%	A*	274.7	14957	0.00481
	Theta*	260.8	918	0.00650
	Block A*	261.8	638	**0.00103**
Random 10%	A*	275.3	15039	0.00489
	Theta*	261.6	4439	0.00417
	Block A*	262.5	845	**0.00140**
Random 20 %	A*	276.4	15351	0.00499
	Theta*	263.3	6229	0.00494
	Block A*	264.3	1159	**0.00185**
Random 30%	A*	277.5	15889	0.00518
	Theta*	265.4	8536	0.00632
	Block A*	266.6	1617	**0.00240**
Random 40%	A*	282.7	18025	0.00584
	Theta*	271.5	12603	0.00904
	Block A*	273.0	2407	**0.00315**
Random 50%	A*	296.9	26146	0.00825
	Theta*	286.2	22721	0.01484
	Block A*	287.8	4476	**0.00468**
Starcraft (random)	A*	300.2	26456	0.01268
	Theta*	285.7	23729	0.11304
	Block A*	286.8	2890	**0.00506**
BG2 (scenarios)	A*	248.7	10796	0.00334
	Theta*	237.2	7043	0.01796
	Block A*	238.0	1034	**0.00147**
DA:0 (scenarios)	A*	409.0	15465	0.00478
	Theta*	392.3	14478	0.02697
	Block A*	393.9	1709	**0.00226**

Table 1: Comparing algorithm performance

of s'' is now set to be s. The line from s to s'' is never longer than the path from s to s' to s''. In Figure 3, let $E5$ be s' whose parent $B1$ is s. When $E5$ is expanded, it will check if its child s'', $E6$, has a straight line to $B1$. If it does, then $E6$'s parent is set to $B1$. In this example, there is no straight line and thus $E6$'s parent is set to $E5$.

5 Experimental Results

We start by comparing the competing algorithms using a 500×500 grid filled with randomly placed obstacles, with the probability of a cell being an obstacle ranging from 0% to 50%. In all our experiments, Block A* used the same 5×5 vertex block LDDB which took 1.2s total to compute. All entries in the top part of Table 1 are averaged over 500 randomly generated obstacle maps, each with 100 path-planning problems based on random start and goal vertices, for a total of 50,000 data points. The bottom part of Table 1 is based on game maps. All results were obtained on a Core 2 2.8 GHz computer with 8 GB of memory.

For low number of obstacles, Theta* expands fewer vertices than A* (top part of Table 1). This helps Theta* outperform A* in run time, despite Theta*'s LOS overhead. As the percentage of obstacles increases, the Euclidean heuristic decreases in effectiveness and Theta* suffers, expanding almost as many nodes as A*. This combined with the LOS overhead causes Theta* to become slower than A*. In contrast, Block A* is always faster than both A* and Theta* (2-3-fold). Furthermore, this gain in speed is not offset by a noticeable increase in path length ("Distance" column); the difference between the path lengths of Theta* and Block A* is at most 0.5%. The number of expansions done by each algorithm ("Expanded" column) needs to be read with care. When A* expands a node, the time cost is roughly constant. When Theta* expands a node, it does an additional LOS call and that cost depends on how obstructed the map is. Block A* does not expand a single cell; it expands a block and the cost varies depending on the block.

Maps with randomly placed obstacles are not necessarily representative of search problems that arise in practice. Hence, we also experimented with real game maps of two different kinds (bottom half of Table 1):

1. RTS (*Starcraft*). These maps were used in the AIIDE 2010 RTS competition (five, sizes 512×512 and 384×512). They have large open areas connected with numerous strategic choke points. This leads to computationally interesting A* behavior as the Euclidean heuristic is far from perfect. The combination of large open areas and a poor heuristic (caused by strategic choke points) is typical in these maps. 50,000 random start/goal pairs were used.

2. Role-playing games (*Baldur's Gate 2* (BG2) and *Dragon Age: Origins* (DA:O)). RPG maps are fundamentally different than RTS maps in that they are maze-like. There are 120 512×512 BG2 maps and 157 DA:O maps varying in size from 49×49 to 1024×1024. BG2 used 93,160 start/goal pairs, stratified by their path lengths, while DA:O used 159,465 start/goal pairs, also stratified.

Theta* is 5-10 times slower than A* for game maps (bottom part of Table 1), but returns shorter paths. In contrast, Block A* is at least 10 times faster than Theta*, while achieving comparable path quality ($\leq 0.5\%$ difference).

Visibility graphs (v-graphs) are commonly used in the games industry (Lozano-Pérez and Wesley 1979). While searching using v-graphs is known to be very fast, they are not used in many commercial games, including *Starcraft*. *Starcraft* can have hundreds of agents/objects being dynamically placed/destroyed/moved on the map in real-time, each leading to a different map. To use v-graphs, each map that has a structural change must be pre-computed (takes over a day). In contrast, a LDDB only needs to be pre-computed once (1.2 seconds) and works for every map.

For graphs with a high percentage of random obstacles, Block A* actually *outperforms* v-graphs (e.g., by almost a factor of two with 40% obstacles). In maps with lots of open spaces, as seen in the game maps, v-graphs are a factor of 3-9 faster at doing an individual search. But it must be kept in mind that v-graphs are effectively only applicable when the map remains static; the pre-processing costs prohibit using them for a highly dynamically changing map.

Why does Block A* perform better than both A* and Theta*? For A* the answer is simple: Block A* benefits from the pre-computed results in LDDB to avoid work. For Theta*, an expensive LOS algorithm is needed during execution. For a LOS algorithm to check a line segment that is M tiles long, it must check at least M tiles, regardless of how efficient the LOS algorithm is. Theta* calculates a LOS for *every* child of the current vertex being expanded.

Figure 4 illustrates the computational overhead of Block A*, A* and Theta*. The darker the diagram, the more com-

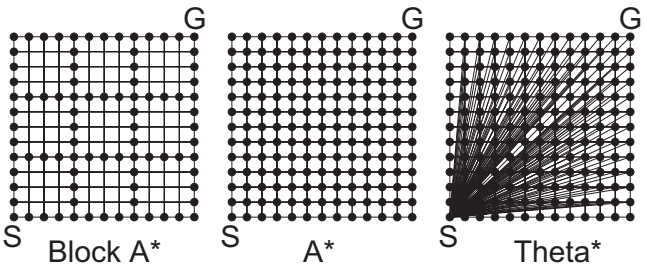

Figure 4: Overheads

Map (size)	Algorithm	Distance	Expanded	Time (s)
1000x1000	A*	518.3	478457	0.38
	Theta*	491.3	478096	22.57
	Block A*	493.1	30032	0.04

Table 2: Using a poor heuristic in a large unobstructed area.

putations. This example uses a 13×13 vertex grid devoid of obstacles and zero heuristic for simplicity. When searching from the start S to the goal G, Block A* with a 5×5 vertex LDDB calculates only the the exterior vertices of the block, denoted by the dark circles on those vertices (left). A* must calculate the g- and h-values for every vertex in the grid (middle). Theta* must expand every vertex done by A* and *in addition* make a LOS check from each child of the current vertex being expanded to S (right).

Lazy Theta* (Nash, Koenig, and Tovey 2010), the most expedient of all Theta* derivatives, calculates a LOS only once for every vertex expanded, at the expense of more expansions. AP Theta* (Daniel et al. 2010) is another derivative that can reduce the number of LOS calls. However, the space and time required for the extra angle maintenance results in a slower average run-times and longer paths compared to Theta*. Although Theta* derivatives can reduce the average number of LOS calls with varying degrees of success, the results are at the expense of longer paths or longer run-times.

Since we are using real numbers for any-angle paths, a heap is used for OPEN. A heap costs $O(NlgN)$ where N is the length of the optimal path. Theta* will perform $O(\sum_{M=1}^{N} MlgM) = O(N^2lgN)$ heap operations. Large open spaces spawn expensive LOS calls as these lines lengthen; this bodes poorly for Theta*, especially for modern game maps (1000×1000 grids).

Table 2 compares the algorithms on a 1000×1000 grid using a zero heuristic with 100 random start/goal pairs. Block A* is fastest, over 560 times faster than Theta* while giving path solutions of comparable quality. While the time performance of Theta* can be improved by a better heuristic the same improvement will also benefit A* and Block A*. Block A* has the benefit of being less sensitive to a bad heuristic compared to A* and Theta*. Given a perfect heuristic, Block A* is still 5-6 times faster than both A* and Theta* (Table 1 with 0% random obstacles). In the absence of a good heuristic, Block A* performs even better.

Path smoothing (PS) (Millington and Funge 2009) is used to remove unnecessary heading changes in a path to reduce its length and make it more realistic. In Table 1 and 2, the maximum difference of 0.5% path length between Theta* and Block A* is reduced to 0.1% with PS.

Finally, Block A* has also been applied to traditional tile (4-way) and octile (8-way) pathfinding on game maps. By changing the LDDB pre-computation, different constraints are encoded into the search. In these experiments (not shown), Block A* was consistently 2-3-fold faster than A*.

6 Conclusions

RTS games remain a challenging domain for path planning algorithms. Firstly, game maps are designed to be interesting, so a good heuristic is hard to find. Block A* performs well with both good and bad heuristics, and is always faster than both A* and Theta*. Secondly, the dynamic nature of games can significantly alter a map. A specialty algorithm good for open areas like AP Theta* but poor for clogged areas will suffer. Theta* may be good for low obstructed areas, but flounders in open areas. A map having open and clogged areas (or can dynamically change) makes deciding which algorithm to use difficult. Block A* is not impacted in these situations as its LDDB contains all pattern possibilities.

From our experiments using real maps used in commercial games, Block A* is consistently faster than both A* and Theta* in both clogged and open areas. The real-time nature of games demand a fast path planning algorithm; in the industry A* is considered too slow. Theta* is always slower than A* per vertex expansion, and much slower than A* when the heuristic is bad, when the map is open or both. In contrast, Block A* is always faster than A*. For all these criteria that make path planning difficult in games—poor heuristics, dynamic maps, and real-time constraints—Block A* is always faster than both Theta* and A*. As well, Block A* will always find shorter and more realistic paths compared to A*; paths comparable to the slower Theta*.

7 Acknowledgments

We thank the reviewers for their comments; and Nathan Sturtevant and Dave Churchill for the game maps. This research was funded by NSERC and iCORE.

References

Daniel, K.; Nash, A.; Koenig, S.; and Felner, A. 2010. Theta*: Any-angle path planning on grids. *JAIR* 39:533–579.

Ferguson, D., and Stentz, A. 2006. Using interpolation to improve path planning: The field D* algorithm. *Journal of Field Robotics* 23(2):79–101.

Lozano-Pérez, T., and Wesley, M. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM* 22(10):560–570.

Millington, I., and Funge, J. 2009. Path smoothing. In *Artificial Intelligence in Games: 2nd Edition*, 251–256.

Nash, A.; Koenig, S.; and Tovey, C. 2010. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *AAAI*.