Watershed Graphs for Faster Path Planning in Binary Occupancy Grids

Patrick Chisan Hew
Defence Science and Technology Group, Australia
Patrick.Hew@defence.gov.au

Abstract

We speed up path planning in binary occupancy grids by using the watershed transform to construct a 'watershed graph' of the terrain's topology, thereby cueing the path planner via a heuristic for the search, setting subgoals, or declaring cells to be blocked. Experiments with A*, Theta*, and Phi* showed that the time invested in constructing the watershed graph can be recovered when repeatedly planning paths through the same terrain, or planning paths over longer distances. Speedup factors exceeding 5× were obtained with only modest inflations in path length. This article will help developers and users of path planners who are looking to reduce runtime without disrupting their algorithm architecture.

1 Introduction

Path planning in binary occupancy grids is keenly relevant to games, robotics, and other applications. Research has sought to model the terrain's global topology, to cue the planner to chokepoints that must be traversed or away from dead ends (Botea, Müller, and Schaeffer 2004), (Björnsson and Halldórsson 2006), (Perkins 2010), (Pochter et al. 2010), (Goldenberg et al. 2010), (Halldórsson and Björnsson 2015), (Uriarte and Ontañón 2016). Previous investigations, however, appear to be unaware of the *watershed transform* from image processing; indeed Björnsson and Halldórsson (2006, 2015) and Pochter et al. (2010) preprocess the binary occupancy grid in a manner that resembles a watershed transform. Perkins (2010) reviewed a number of possibilities and selected Voronoi diagrams for his Brood War Terrain Analyzer. But to compute a Voronoi diagram, the accessible terrain has to be vectorized; that is, taking the terrain that is represented as cells that are accessible vs blocked and inferring the existence of polygonal regions that are accessible vs blocked (see for example step 1 of Perkins 2010 and Uriarte and Ontañón 2016). In contrast, a watershed transform operates directly on the cell-based representation of the terrain.

This article investigates how a watershed transform can be used to infer a terrain's global topology, and the net reduction in runtime for path planning that can be achieved. The intent is to help developers and users of path planners with a method that they could apply as a preprocessor, without disrupting the rest of their software. To this end, the proposal is tested on A* for grid paths and Theta* (Nash, Koenig, and Tovey 2010) and Phi* (Nash, Koenig, and Likhachev 2009) for short any-angled paths. While newer and faster any-angled path finders exist (Harabor et al. 2016), (Cui, Harabor, and Grastien 2017), the older algorithms remain in use due to low conceptual cost of entry and high cost of change. If the watershed graph helps A* then it should help planners based on it.

2 Constructing a Watershed Graph

We recall some terminology from digital image processing: A *blob* is a collection of cells that are connected (it is a connected component in the grid graph. We use 'blob' for brevity.) A *4-blob* consists of cells that touch at edges (they are *4-connected*). Meanwhile an *8-blob* consists of cells that touch at edges or corners (they are *8-connected*).

The following text will provide full details on constructing a watershed graph. In overview (Figure 1):

1. Focus on the largest 8-blob of accessible cells.
2. Calculate the distance transform, then apply a watershed transform to the complement.
3. Obtain the ridges.
4. Obtain transits across ridges.
5. Grow the basins into a segmentation of the cells.
6. Initialize graph from transits.
7. Add goal.
8. Add edges from nodes in the same segment.
9. Estimate distances from goal to all nodes.

The output is:

- A weighted undirected graph $\mathcal{G}$, called the *watershed graph*, that models the terrain's global topology.
- A function $\widehat{D}(y)$ that returns the distance to the goal via $\mathcal{G}$ from any node $y \in \mathcal{G}$.
- A segmentation S of the accessible cells that maps them into the global topology, and an associated function $S(x)$ that returns a unique segment for a given cell corner x.

Steps 1–6 are performed whenever the terrain map changes. Steps 7–9 are then performed whenever the goal changes.

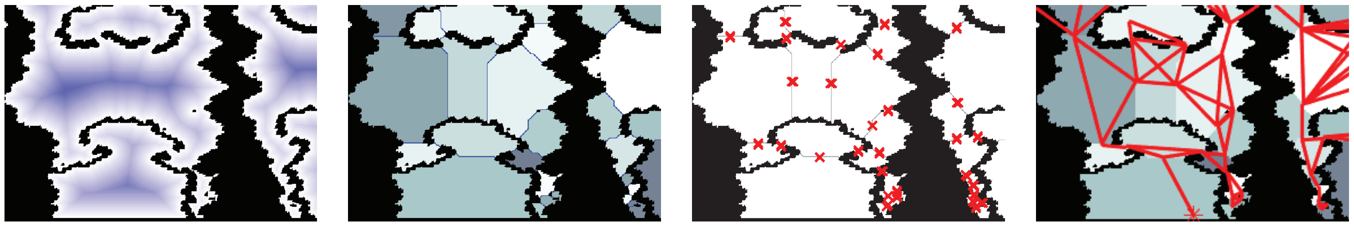

(a) Distance transform (b) Basins and ridges (c) Transits across ridges (d) Segments and graph

Figure 1: Constructing a watershed graph. a) Focus on the largest 8-blob of accessible cells, distance transform (step 1–2). b) Watershed transform of complement of distance transform to identify basins and ridges (step 3). c) Obtain transits across ridges (steps 4). d) Grow the basins into a segmentation of the accessible cells, initialize graph from the transits, add goal, add edges from nodes in the same segment, estimate distances (steps 6–9).

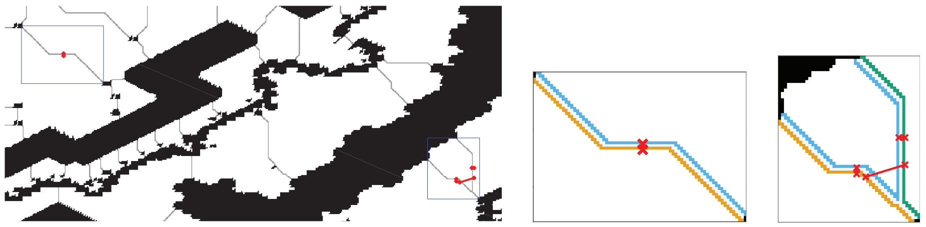

(a) (b)

Figure 2: Obtaining transits across ridges. The main picture shows the ridges, and two cases of enclosing a rectangle R' around a ridge R. The close-up pictures show A' for each basin A that abuts R (A' consists of the cells in $A \cap R'$ that share a corner with a cell in R); the colors correspond to different basins. Transits are shown as line segments with '$\times$' on each end: For each pair of basins A, B the transit (a, b) from A to B is the tuple of the cell corners $a \in A'$, $b \in B'$ that minimize the round-trip Euclidean distance from x to a to b back to x given x is the centroid of $A' \cup B'$. a) Usual case of a ridge that has two abutting basins. b) An atypical case that can arise out of the watershed transform, but transits are still obtained.

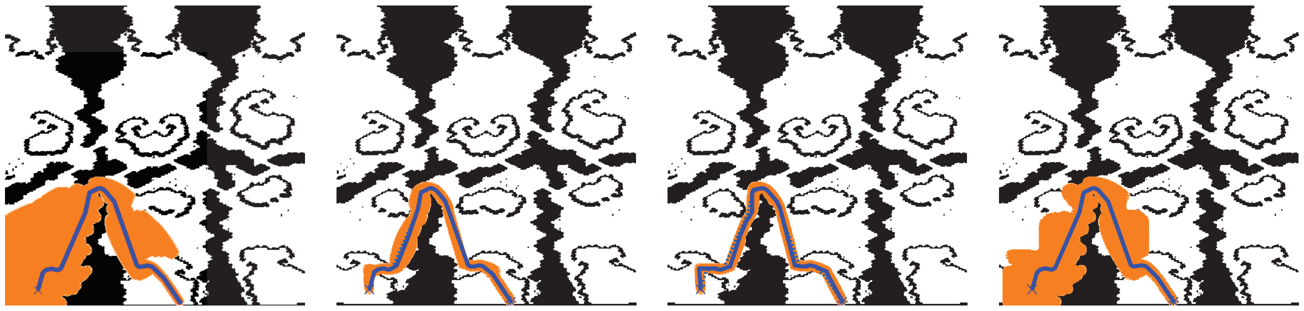

(a) Euclidean Heuristic (b) Watershed Graph Heuristic (c) Watershed Graph for Subgoals (d) Watershed Graph to Block Cells

Figure 3: Applying the watershed graph using Lazy Theta* (Nash, Koenig, and Tovey 2010) as the path planner. The start is at bottom-left, the goal is at bottom-middle. The shaded region shows the nodes that were open or closed when path planning ended. a) Path planning guided by the Euclidean heuristic. b) Path planning guided by the watershed graph heuristic (Section 3.1). The planner inspects fewer nodes and hence arrives at a path more quickly, at the cost of a slight inflation in path length. c) Path planning under subgoals set by the watershed graph (Section 3.2). The planner inspects even fewer nodes and is thus even faster, but the resulting path has been forced to pass through the subgoals, increasing its length. d) Path planning guided by the Euclidean heuristic where cells are declared blocked due to the watershed graph (Section 3.3). The planner inspects fewer nodes than in the original grid, with a small inflation in path length.

2.1 Restrict to Largest 8-Blob of Accessible Cells

Given a binary occupancy grid, we restrict our attention to the largest 8-blob of accessible cells. For two cell corners in an 8-blob named *start* and *goal*, a path planner will return a path from the start to the goal that stays within the blob.

Our use of 8-blobs makes tacit assumptions about the vehicle for which a path is being planned, namely: the vehicle moves at the same speed in any direction, and it is small and nimble enough to 'squeeze through the diagonal' between cells that touch at corners but not faces.

2.2 Calculate the Distance Transform, Apply a Watershed Transform to the Complement

We obtain the Euclidean distance transform E of the accessible cells: $E(i, j)$ is the Euclidean distance of cell (i, j) to its nearest blocked cell. Next, we get the complement $\tilde{E}$ of the distance transform, namely $\tilde{E}(i, j) = \max E - E(i, j)$ for each cell (i, j). We then apply Meyer's (1994) watershed transform to $\tilde{E}$. This variant of the watershed transform takes a grayscale image as its input and treats it as the height map of some terrain. One imagines placing sources of water at the terrain's local minima. The water will pool as *basins*, and as the basins expand they will meet at *ridges*. In practice, the basins are grown by repeated dilation of the local minima by a disc.

The core idea is that ridges can arise at chokepoints in the terrain: we are acting on the complement of the distance transform so the basins start at maximum distance from the blocked cells, hence the ridges form in accessible cells that are near to blocked ones. Other processing (Meyer 1994), (Eddins 2002) can prevent basins that are too small, although the author found this refinement was unnecessary.

2.3 Obtain the Ridges

As noted, the ridges can be caused by chokepoints in the terrain. We segment the ridges' cells into 4-blobs. Each 4-blob will henceforth be referred to as a *ridge*.

2.4 Obtain Transits Across Ridges

We know that the basins are 8-connected and that the only way from one basin to another is via a ridge. We now build *transits* between basins so that we can tell the planner how to leave a basin and where it can go next. Let R be a ridge. A basin *abuts* R if at least one of the basin's cells is 8-connected to a cell in R. To build *the transits at* R (Figure 2):

1. Let R' be the rectangle that encloses R with a margin of one cell on all sides.
2. Iterate over the cells of R to find the basins that abut R.
3. For each pair of basins A, B that abut R:
   (a) Let A' be the cells in $A \cap R'$ that share a corner with a cell in R. Likewise let B' be the cells in $B \cap R'$ that share a corner with a cell in R.
   (b) Let x be the centroid of $A' \cup B'$.
   (c) Obtain $a \in A'$, $b \in B'$ that minimize $|xa| + |ab| + |bx|$, the round-trip Euclidean distance from x to a to b back to x. The tuple of cell corners (a, b) is declared to be the *transit from* A *to* B *at* R.

We collect the transits at all ridges. A ridge may have no transits, and the process given above handles this situation. (Consider a blocked region that has a crinkled boundary with small pockets of accessible cells. The cells inside a pocket can be assigned to a ridge that has no basins abutting it.) The construction of R' and hence A', B' saves time: if a cell touches R then it must be in R', and then the minimization of round-trip distance only considers those cells.

2.5 Grow Basins into a Segmentation of the Cells

We now grow the basins into a segmentation of the accessible cells. Hence for any given cell corner, we can get a segment that contains the corner, the transits from that segment, and thus (by step 4) guidance on where to go next.

Algorithm 1 performs the segmentation. It starts with cells that have been assigned to segments as they are in basins, and cells that are unassigned as they are in ridges. It then iteratively looks for cells that are unassigned but 8-connected to segments, and reassigns them to segments. In effect, the segments absorb the ridges in a manner akin to a flood fill. As the accessible cells form a single 8-blob (step 1), every accessible cell will eventually be assigned to a segment.

Algorithm 1: Grow the basins to absorb the ridges so that every accessible cell is assigned to a segment.

```
1  $S \leftarrow$ SegmentsFromBasins($B$, $M$)
       input  : $B$ labels the basins: $B(i, j)$ is the basin
                containing cell $(i, j)$. If $B(i, j) = 0$ then
                cell $(i, j)$ is blocked or in a ridge.
       input  : $M$ is the binary occupancy grid.
       output: $S$ labels the segments: $S(i, j)$ is the
                segment containing cell $(i, j)$.
2      $S \leftarrow B$     ˙Grow the segmentation from the basins.
3      $R \leftarrow (S = 0)$ and (not $M$)
4      while any $R$ do
           ˙Get unassigned cells that touch segments.
5          $N \leftarrow$ MaxLabelsTouching($R$, $S$)
           ˙Reassign those cells to segments.
6          $S(N > 0) \leftarrow N(N > 0)$
7          $R \leftarrow (S = 0)$ and (not $M$)
8  $N \leftarrow$ MaxLabelsTouching($BW$, $L$)
9      $N \leftarrow$ Zero matrix the same size as $BW$
10     foreach $(i, j)$ such that $BW(i, j)$ is true do
11         $N(i, j) \leftarrow \max\{L(i', j')$ :
             $(i', j')$ is 8-connected to $(i, j)\}$
```

2.6 Initialize Graph From Transits

At step 4, we obtained transits across the ridges. We now construct the *initial watershed graph* as the undirected graph that takes as its nodes the transits' cell corners, from all of the ridges. To emphasize, each transit generates two nodes in the initial watershed graph. Moreover nodes a, b are declared as adjacent in the initial watershed graph if the tuple of cell corners (a, b) is a transit (in the sense defined at step 4).

2.7 Add Goal

The previous steps are performed whenever the map of the terrain is changed. Indeed if the map never changes then the initial watershed graph only needs to be constructed once.

We now perform the steps for building a *watershed graph given goal* $\mathcal{G}$ (abbreviated to *watershed graph*). The first of these steps is to declare that the goal is a node in $\mathcal{G}$. In practice, one adds the goal to $\mathcal{G}$ *unless* it is already there. The latter situation arises if the goal is also an element in a transit. (Note that the goal is a cell corner, as per step 1.)

2.8 Add Edges From Nodes In the Same Segment

We now complete the idea that we started at step 4. Suppose that the path planner has entered a segment. If the segment contains the goal then the planner should move to the goal. Otherwise it should move to a transit that leaves the segment.

Formally, let x be a cell corner and S be a segmentation of the accessible cells. We construct the following function to map cell corners to segment labels

$$S(x) \triangleq \max\{S(i,j) : x \text{ is a corner of cell } (i,j)\}$$

(In general, the maximum is taken over the four cells that have x as a corner.) Now given the segmentation S obtained at step 5, declare that nodes a, b are adjacent in the watershed graph $\mathcal{G}$ if $S(a) = S(b)$.

2.9 Estimate Distances from Goal to All Nodes

We now construct a function $\widehat{D}(y)$ that estimates the distance to the goal from any node $y \in \mathcal{G}$. Choose a function $\widehat{d}(x,y)$ that estimates the distance *in the binary occupancy grid* from x to y. For each pair of adjacent nodes a, b in $\mathcal{G}$, set the weight on the edge from a to b as $\widehat{d}(a,b)$. Then apply Dijkstra's algorithm to $\mathcal{G}$ to construct $\widehat{D}(y)$ as the distance *via the graph* $\mathcal{G}$ from the goal to all nodes $y \in \mathcal{G}$.

As one choice, $\widehat{d}(x,y)$ could be declared as the Euclidean distance from x to y. More accurate estimates are possible – for example, an A* grid search – but may take longer.

3 Applying a Watershed Graph

In the previous section, we constructed a graph $\mathcal{G}$ that models the terrain's global topology from watershed analysis and chose a function $\widehat{d}(x,y)$ that estimates the distance from x to y (for example, the Euclidean distance). The result was a function $\widehat{D}(y)$ that estimates the distance to the goal from any node $y \in \mathcal{G}$, a segmentation S of the accessible cells that maps them into the global topology, and a function $S(x)$ that returns a unique segment for a given cell corner x. We now apply those calculations to guide path planning (Figure 3).

For any given cell corner x, we get the segment $S(x)$, get the exits y from that segment, and use the path from y to goal *in the watershed graph* $\mathcal{G}$ to guide a path in the binary occupancy grid. The guidance is provided as a heuristic, by setting subgoals, or marking cells as being blocked. None of these ideas are new, and precedents in the literature are noted in the text that follows, but we put them in the context of the watershed graph for completeness.

3.1 Watershed Graph Heuristic

For any cell corner x the *watershed graph heuristic* $\mathtt{h}(x)$ is

$$\mathtt{h}(x) \triangleq \min_{z\in\mathcal{G}}\{\widehat{d}(x,z) + \widehat{D}(z) : S(x) = S(z)\}$$

That is, retrieve the segment that contains x, get the exits z from that segment, and choose the exit that has the shortest estimated path to the goal. The resulting heuristic is then used by the path planner, as usual. The watershed graph heuristic is conceptually the same as the Gateway Heuristic (Björnsson and Halldórsson 2006) noting that they used A* grid searches to obtain their distance estimates $\widehat{d}(x,y)$. It likewise resembles the Portal Heuristic (Goldenberg et al. 2010) though they replaced $\widehat{d}(x,y)$ with the true distance (in the binary occupancy grid) from x to y.

3.2 Watershed Graph Subgoals

Given a start x, we set subgoals that will lead the planner to the goal. Let $\mathcal{P}$ be a path planner: for cell corners x, y, $\mathcal{P}$ will return a short path $\mathcal{P}(x,y)$ from x to y. Then:

1. Choose

$$y = \arg\min_{z\in\mathcal{G}}\{\widehat{d}(x,z) + \widehat{D}(z) : S(x) = S(z)\}$$

   That is, retrieve the segment that contains x, look at the exits z from that segment, and set y as the exit that has the shortest estimated path to the goal.

2. Let $y = y_0, y_1, \dots y_{n-1}, y_n =$ goal be nodes from y such that $\widehat{D}(y) = \sum_{k=1}^{n}\widehat{d}(y_{k-1}, y_k)$. Such a sequence arises when calculating $\widehat{D}(y)$ via Dijkstra's algorithm (step 9).

3. Declare $\{y_k\}_k$ to be the *watershed graph subgoals*. Correspondingly, we plan the path from x to goal as

$$\text{Path from } x \text{ to goal} \triangleq \bigcup_{k=1}^{n} \mathcal{P}(y_{k-1}, y_k)$$

Step 3 is pleasingly parallel — a path from y_{k-1} to y_k can be obtained for each k without reference to the paths for any other k. For this article, we perform step 3 sequentially. Subgoals have been studied extensively; see for example Uras and Koenig (2013, 2015).

3.3 Watershed Graph Cell Blocking

Given a start x, we redeclare cells in the binary occupancy grid as being blocked and use the resulting grid to plan a path from x to the goal. Specifically:

1. Construct $Y = \{y_k\}_k$ as the watershed graph subgoals (see Section 3.2).
2. Let $\mathcal{S}_Y$ be the cells of the segments that contain Y. Let $\mathcal{R}$ be the ridge cells. Construct the binary occupancy grid that has $\mathcal{S}_Y \cup \mathcal{R}$ as its accessible cells and use it to plan the path from x to the goal.

Cell blocking was applied to dead-ends by Björnsson and Halldórsson's (2006) and swamps by Pochter et al. (2010).

4 Experiments

4.1 Method

The hypothesis was that the watershed graph would yield a net reduction in runtime, though possibly with an increase in path length. The hypothesis was tested using the three proposed applications of the watershed graph (Heuristic, Subgoals, Cell Blocking), on the Moving AI 2D path planning benchmarks (Sturtevant 2012), under three path planners:

- A* Grid (A* applied to planning a grid path) as it is the foundation for many other search algorithms.
- Lazy Theta* (Nash, Koenig, and Tovey 2010) as a representative any-angled path planner.
- Phi* (Nash, Koenig, and Likhachev 2009) as it is a component in dynamic path planning at any angle.

The open nodes were stored on a binary min heap with tiebreaking on the calculated distances from the start (key on $\mathtt{f}$-scores, tiebreak on $\mathtt{g}$-scores). Distance estimates $\widehat{d}(x,y)$ were made by using the Euclidean distance from x to y.

Steps were taken to ensure that reductions in runtime were genuinely due to the watershed graph, and not merely due to an unfair substitution of just-in-time compiled code with precompiled code. The point arose from the author's use of the MATLAB Image Processing Toolbox, as the watershed transform and blob processing routines used precompiled binaries. While the path planners were implemented in MATLAB, the line-of-sight testing, upheaping, and downheaping were implemented in C and called as precompiled binaries (mex-functions). Profiling indicated that the line-of-sight testing, upheaping, and downheaping accounted for almost all of the runtime in path planning (over 80 percent and often in excess of 90 percent). Hence we are validly trading precompiled code for precompiled code.

The experiments were conducted on an Intel i7-7700 CPU running at up to 3.60 GHz (Dell OptiPlex 7050 desktop computer), with 4 cores supplying 8 logical processors. The trials were managed using the MATLAB Parallel Processing Toolbox with 6 workers, so the CPU was working at about 90 percent capacity. Runtimes were measured using wall-clock time following best practices (McKeeman 2016).

The measures for testing the hypothesis were:

- *Speedup factor* calculated as

$$\text{Speedup Factor} = \frac{\text{Runtime using Euclidean Heuristic}}{\text{Runtime using Watershed Graph}}$$

  Speedup factors greater than one correspond to the watershed graph reducing runtime. Creating and exploiting the watershed graph takes time, but the hypothesis was that the investment would be recouped when planning the path. Moreover the speedup factor should increase as the distance from start to goal increases, as a longer path takes longer to plan. The watershed graph methods were further anticipated to be faster on the 'game', 'street', and 'room' maps but slower on the 'maze' and 'random' maps. The 'game', 'street' and 'room' maps have a global topology that can be exploited: the terrain has rooms that can trap the planner and force it to backtrack, and chokepoints that the planner should be cued to. The 'maze' maps only have these properties when the corridors are wide, and 'random' maps lack them entirely.

- *Path length inflation* is a percentage calculated as

$$\text{Path Length Inflation} = \left(\frac{\text{Path Length using Watershed Graph}}{\text{Path Length using Euclidean Heuristic}} - 1\right) \times 100\%$$

  Desirably, the path length inflation would be as small as possible. In emphasizing the global topology over the local topology, it was anticipated that the watershed graph would distort the paths that would otherwise be found. Indeed the greatest distortion was anticipated from setting subgoals as it forces the paths to traverse the transits. Distortion was likewise anticipated when the watershed graph was used as a heuristic as it weights the paths to traverse the transits. Cell blocking was anticipated to cause distortion only in those rare cases where the actual shortest path was homotopically different from the one estimated from the watershed graph (that is, the actual shortest path went through a different set of rooms or chokepoints than the one predicted by the watershed graph).

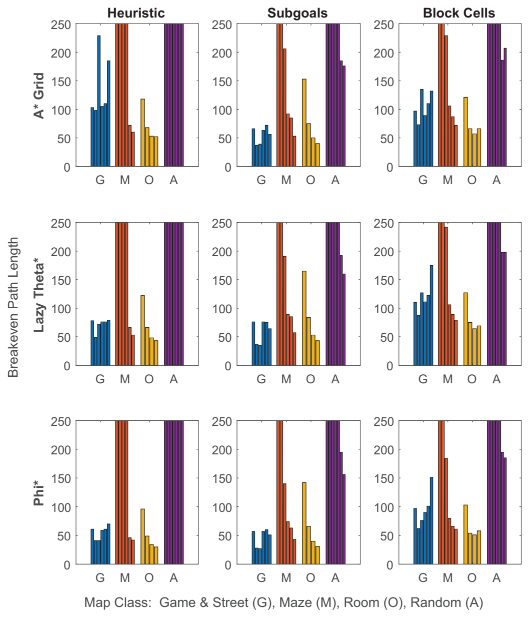
We test the worst-case situation for constructing and applying a watershed graph in performing all processes (Section 2, steps 1–9) from an uninitialized state, every time. If multiple queries were being made about the same map, there would of course be considerable savings in time by storing the output of processes that need only be performed once per map (steps 1–6) or once per goal (steps 7–9).

4.2 Results

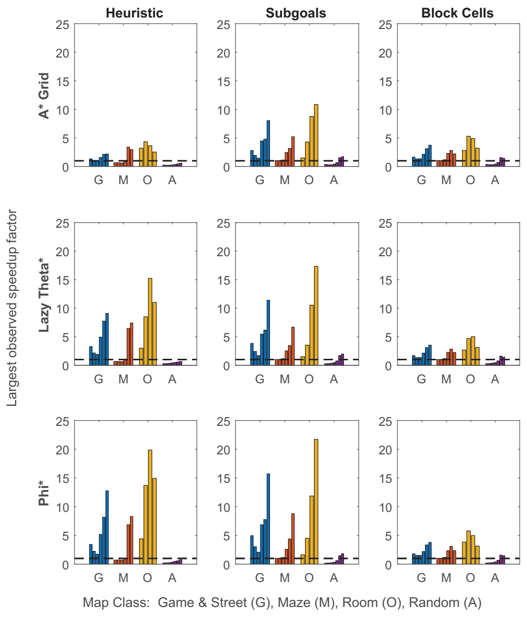
Figures 4.a and 4.b show the results for speedup factor. Figure 4.a establishes whether the time invested in the watershed graph is actually recovered by charting the *break even path length*, the path length in the benchmark set when the speedup factor first exceeded one. The results were largely as anticipated. Under all three applications of the watershed graph and across all three of the path planners, the speedup factor eventually exceeded one on the 'game', 'street', and 'room' maps. The path length when this occurred (break even) was on the order of 50-150 cells, noting that the maps were 512×512 cells. The speedup factor did not reach one on 'maze' maps when corridors were thin, nor on 'random' maps when the fractions of blocked cells were low.

Speedup factors generally increased as path length increased. Figure 4.b shows the largest speedup factors that were observed. Factors on the order of 5-15× were observed in favourable cases, namely 'sc1', 'street', and 'room' maps. An unanticipated behaviour was that the speedup factor could *decrease* as path lengths increased; the behaviour showed up in the 'game' map sets 'bg512', 'dao', 'da2', and in 'room' and 'maze' maps. The likely cause is that very long paths achieve their length by traversing large portions of the map, so path planning explores the entire map. Indeed the 'bg512', 'dao', and 'da2' maps consist of '1D' chains of rooms, versus '2D' terrain such as the 'sc1' maps.

Figure 4.c shows the results for path length inflation. The results were again as anticipated. Across all three of the

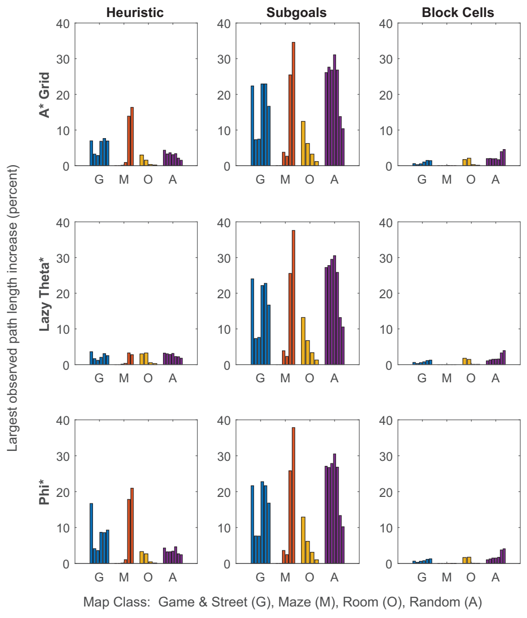


(a) Break even path length



(b) Largest observed speedup factors

Figure 4: Results for each algorithm (A* Grid, Lazy Theta*, Phi*), across the applications of the watershed graph (Heuristic, Subgoals, Block Cells), and map classes (Game & Street, Maze, Room, Random). (a) Break even path length (path length when speedup factor first exceeded one). Smaller values are better. (b) Largest observed speedup factors. The dashed line marks a factor of one – larger speedup factors are better.



(c) Largest observed path length inflations

Figure 4: Results continued. (c) Largest observed path length inflations. Smaller inflation is better.

path planners, setting subgoals caused the greatest inflation, on the order of 20 percent. The heuristic suffered the next-largest inflation, on the order of 5 percent. Cell blocking suffered no inflation except in rare cases. Path length inflation was especially acute on 'maze' maps and increased with corridor width. We can account for this behaviour as follows: when obtaining the transit across a ridge (step 4 in building the watershed graph), the transit's location is weighted towards the centroid of the ridge. Hence in mazes, the transits tend towards the middle of corridors. But in a shortest path, changes in direction are tight around corners.

5 Conclusion

Obtaining the watershed graph can speed up path planning if the terrain contains rooms and chokepoints, and the paths being obtained are long but do not fill the map. Hence watershed graphs could be helpful in interior, underground, or urban terrain but less helpful in forests. On favourable terrains, speedup factors on the order of 5-15× were achieved with A*, Theta*, and Phi*. Inflations in path lengths were modest, and were largely due to a biasing of paths towards the middle of corridors. While the research was performed using MATLAB, the requisite digital image processing algorithms may be found elsewhere; for example in OpenCV.

References

Björnsson, Y., and Halldórsson, K. 2006. Improved Heuristics for Optimal Pathfinding on Game Maps. In *Proceedings* *of the Second AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-06)*, 9–14.

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

Cui, M.; Harabor, D.; and Grastien, A. 2017. Compromise-free pathfinding on a navigation mesh. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 496–502.

Eddins, S. 2002. The watershed transform: Strategies for image segmentation. The MathWorks: Technical Articles and Newsletters. https://au.mathworks.com/company/newsletters/articles/the-watershed-transform-strategies-for-image-segmentation.html, Accessed 2018-08-16.

Goldenberg, M.; Felner, A.; Sturtevant, N.; and Schaeffer, J. 2010. Portal-Based True-Distance Heuristics for Path Finding. In *Proceedings of the Third Annual Symposium on Combinatorial Search (SOCS-10)*, 39–45.

Halldórsson, K., and Björnsson, Y. 2015. Automated Decomposition of Game Maps. In *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)*, 122–127.

Harabor, D.; Grastien, A.; Öz, D.; and Aksakalli, V. 2016. Optimal any-angle pathfinding in practice. *Journal of Artificial Intelligence Research* 56:89–118.

McKeeman, B. 2016. MATLAB Performance Measurement. http://au.mathworks.com/matlabcentral/fileexchange/18510-matlab-performance-measurement. (Accessed 2017-02-21).

Meyer, F. 1994. Topographic distance and watershed lines. *Signal Processing* 38(1):113 – 125. Mathematical Morphology and its Applications to Signal Processing.

Nash, A.; Koenig, S.; and Likhachev, M. 2009. Incremental Phi*: Incremental Any-Angle Path Planning on Grids. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1824–1830.

Nash, A.; Koenig, S.; and Tovey, C. 2010. Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. In *Proceedings of the AAAI Conference on Artificial Intelligence AAAI*.

Perkins, L. 2010. Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-10)*, 168–173.

Pochter, N.; Zohar, A.; Rosenschein, J. S.; and Felner, A. 2010. Search Space Reduction Using Swamp Hierarchies. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, 155–160.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.

Uras, T., and Koenig, S. 2015. Speeding-up Any-Angle Path-Planning on Grids. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Uriarte, A., and Ontañón, S. 2016. Improving Terrain Analysis and Applications to RTS Game AI. In *Artificial Intelligence in Adversarial Games: Papers from the AIIDE Workshop, AAAI Technical Report WS-16-21*, 15–20.