Edge N-Level Sparse Visibility Graphs: Fast Optimal Any-Angle Pathfinding Using Hierarchical Taut Paths

Shunhao Oh, Hon Wai Leong
Department of Computer Science
National University of Singapore
ohoh@u.nus.edu, leonghw@comp.nus.edu.sg

Abstract

In the Any-Angle Pathfinding problem, the goal is to find the shortest path between a pair of vertices on a uniform square grid, that is not constrained to any fixed number of possible directions over the grid. Visibility Graphs are a known optimal algorithm for solving the problem with the use of pre-processing. However, Visibility Graphs are known to perform poorly in terms of running time, especially on large, complex maps. In this paper, we introduce two improvements over the Visibility Graph Algorithm to compute optimal paths. Sparse Visibility Graphs (SVGs) are constructed by pruning unnecessary edges from the original Visibility Graph. Edge N-Level Sparse Visibility Graphs (ENLSVGs) is a hierarchical SVG built by iteratively pruning non-taut paths. We also introduce Line-of-Sight Scans, a faster algorithm for building Visibility Graphs over a grid. SVGs run much faster than Visibility Graphs by reducing the average vertex degree. ENLSVGs, a hierarchical algorithm, improves this further, especially on larger maps, with millisecond runtimes even on 6000×6000 maps. On large maps, with the use of pre-processing, these algorithms are at least an order of magnitude faster than existing algorithms like Visibility Graphs, Anya and Theta*.

Introduction

In many pathfinding applications involving open spaces, it is common strategy to abstract a 2D map into a uniform square grid (Algfoor, Sunar, and Kolivand 2015). Many grid-based pathfinding algorithms are 8-directional, where the agent can only move in the four cardinal and four diagonal directions along the grid. We consider the Any-Angle Pathfinding problem, where this constraint is removed. The start and goal are vertices of the grid. The objective is to compute the shortest path in terms of Euclidean distance from the start to the goal that does not intersect any blocked tiles in the grid.

There are many algorithms for optimal 8-directional pathfinding, like a simple 8-directional A*, or faster algorithms like Jump-Point Search (Harabor and Grastien 2011) and Subgoal Graphs (Uras, Koenig, and Hernández 2013). On the other hand, computing optimal any-angle paths is more difficult. Thus, many existing Any-Angle Pathfinding algorithms like Theta* (Nash et al. 2007) and Block A* (Yap et al. 2011), are heuristic in nature.

A known optimal Any-Angle Pathfinding algorithm is A* on Visibility Graphs (Lozano-Prez and Wesley 1979). However, Visibility Graphs can be inefficient in practice for two reasons. Firstly, Visibility Graph construction requires many Line-of-Sight Checks, quadratic on number of tiles in the grid. While this can be partially solved by pre-processing the visibility graph, a second issue is the high average vertex degree, slowing down an A* search on the graph.

Two recent Any-Angle Pathfinding algorithms have also been shown to perform significantly better than Theta* in practice. The first algorithm is Anya as described in (Harabor et al. 2016), a fast online optimal algorithm based on searching intervals instead of vertices. The second algorithm, in (Shah and Gupta 2016), describes multiple optimisations to speed up A* on a Visibility Graph over a quadtree. We refer to this algorithm as SG16. These algorithms have the advantage of being online, requiring no pre-processing.

In this paper, we introduce two improvements to the Visibility Graph algorithm, Sparse Visibility Graphs (SVGs) and Edge N-Level Sparse Visibility Graphs (ENLSVGs), which are at least an order of magnitude faster than existing algorithms like Theta*, Anya16 and SG16. The relationship to other algorithms can be found in (Uras and Koenig 2015a). SVGs are constructed from removing unnecessary edges from the Visibility Graph. ENLSVGs are constructed by building a hierarchy over an underlying SVG.

The SVG and ENLSVG algorithms are fast and optimal, but are offline algorithms, using a slower pre-computation step so that many shortest path queries can be made quickly. A drawback of offline algorithms is that the pre-computation step needs to be repeated each time the map changes.

Both algorithms, SVGs and ENLSVGs, are based only on the simple concept of pruning non-taut paths to reduce the search space. Through these algorithms, we observe the relationship between taut and optimal paths. Optimal paths are difficult to compute in general, but taut paths, being locally optimal rather than globally optimal, can be computed very easily in constant time. We show how just simple taut path restrictions can greatly reduce the search space for an optimal search. Pruning taut paths on one level forms SVGs, and extending it to n levels of pruning forms ENLSVGs.

Previous work making use of taut paths in Any-Angle search include Anya (Harabor and Grastien 2013) and Strict Theta* (Oh and Leong 2016). The idea of building a multi-

level hierarchy for optimal pathfinding is based on previous work on N-Level Subgoal Graphs (Uras and Koenig 2015b). N-Level Subgoal Graphs prune vertices using shortest paths, while ENLSVGs prune edges using taut paths.

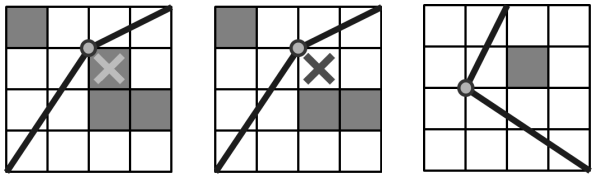
We also introduce Line-of-Sight Scans, a fast algorithm for querying visible neighbours of a vertex in the grid. This replaces Line-of-Sight Checks for building the Visibility Graph and inserting the start and goal points into the graph.

Preliminaries

As previously mentioned, A* on Visibility Graphs (VGs) returns optimal any-angle paths. The vertices of a Visibility Graph consist of the start and goal vertices, and all convex corners of obstacles. We connect all pairs of vertices with Line-of-Sight. One method to build a Visibility Graph is to run Line-of-Sight Checks between every pair of vertices on the grid using Bresenham's line-drawing algorithm (Bresenham 1965). The Rotational Plane Sweep Algorithm (Choset 2005) can also be used to construct a Visibility Graph in $O(n^2 \log n)$ time on a set of polygons with total n vertices. Both of these methods can be very slow on large grids. Thus, it is more reasonable to pre-process a Visibility Graph on the grid, and reuse the graph for multiple shortest-path queries.

As start and goal vertices differ for each query, we leave them out of the pre-processed graph. During a shortest-path query, we connect the start and goal vertices to the Visibility Graph by doing Line-of-Sight Checks to all other existing vertices. We remove them after the query. To take care of the special case where there is Line-of-Sight between the start and goal, we do an initial Line-of-Sight check for a direct path before attempting to add them to the visibility graph.

We use taut path restrictions to reduce the search space. Informally, a taut path is a path which, when treated as a string, cannot be made "tighter" by pulling on its ends. Formally and practically, a path is taut if and only if every heading change in the path wraps tightly around some obstacle (Oh and Leong 2016). As shown in Figure 1, only a single obstacle needs to be checked per heading change to determine if a path is taut (Figure 1c can never be taut). As all optimal paths are taut, if we restrict the search space to taut paths, the optimal path will be included in the search space.



(a) (b) (c)

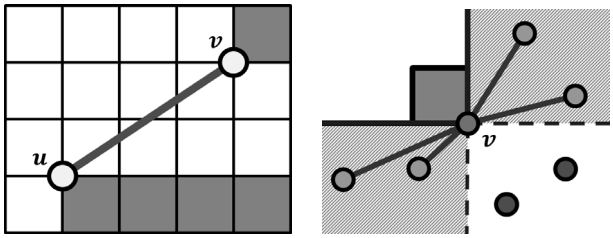
Figure 1: (a) is taut, while (b) and (c) are not.

For the rest of this paper, we make use of Taut A* for graph search in place of the standard A* algorithm. In Taut A*, whenever we attempt to generate a successor v from the current state u, we first check for tautness. v can be a successor of u only if the subpath $parent(u) - u - v$ is taut. We use the Euclidean distance to the goal as our heuristic.

Sparse Visibility Graphs

Many edges in the Visibility Graph are unnecessary, as they are never used in the final path found by A*. In particular, these edges cannot be part of any taut path between any pair of start or goal points, unless the start or goal is one of the edge's endpoints, in which case the edge will be added anyway when connecting the start or goal to the visibility graph.

Refer to the edge uv in Figure 2a. Suppose endpoint v is neither the start nor the goal. Thus, in any path involving edge uv, the path must leave v to move to another vertex. However, in all legal directions the path can leave v from, the path will not be taut, and thus not optimal. We say that the edge uv has no taut exit from v.



(a) Edge with no taut exit from v (b) Taut regions (shaded) of v

Figure 2: Edges without taut exit directions are unnecessary.

To identify the edges to be pruned, we consider the taut regions around each vertex v in the graph. A vertex u is in the taut region of vertex v if the edge uv has a taut exit from v. To find the taut regions, we need only consider the obstacles adjacent to v as shown in Figure 2b. We prune any edge uv where any one of the endpoints does not lie within the taut region of the other endpoint (Figure 3). The remaining edges make up the Sparse Visibility Graph (SVG).

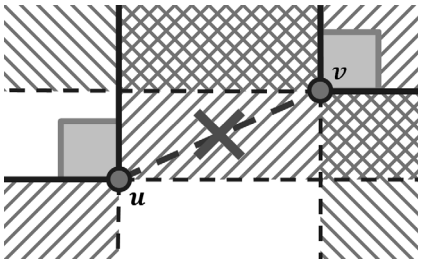


Figure 3: v lies within the taut region of u, but u does not lie within the taut region of v. Edge uv is pruned.

Collinear Points

We describe our policy on collinear points in an SVG. Naively, a set of k collinear points would form a size k clique due to Line-of-Sight between any two points in the set (Figure 4). This is clearly wasteful and unnecessarily increases the average vertex degree. In these cases, it suffices for each vertex to have edges only to its closest neighbour on each side of the point on the line. Intuitively, we can imagine each vertex as being an epsilon-size Line-of-Sight obstruction.

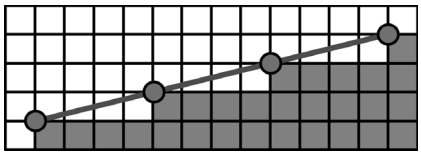


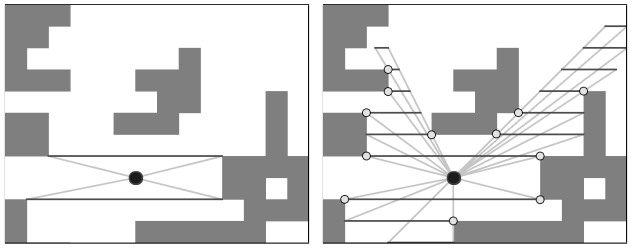
Figure 4: Every pair of points on the line has Line-of-Sight.

Fast Construction Using Line-of-Sight Scans

Constructing Visibility Graphs using Line-of-Sight checks between every pair of vertices takes $\Theta(V^2)$ Line-of-Sight Checks even in the best case. This is because the computation is non-local. Even on dense maps where Line-of-Sight is uncommon, Line-of-Sight Checks are still conducted between vertices on opposite ends of the map.

In place of individual vertex-to-vertex Line-of-Sight Checks, we introduce Line-of-Sight Scans, which computes the set of visible vertices from a single vertex. Intuitively, a Line-of-Sight Scan from a vertex v is a radial outwards scan which breaks whenever it hits an obstacle. We implement this using a similar method to the interval search used by Anya (Harabor and Grastien 2013).

The key advantage of Line-of-Sight Scans is that it is local. For each vertex, the running time of Line-of-Sight Scans depends on the number of visible vertices, while Line-of-Sight Checks depends on the total number of vertices in the entire map. Line-of-Sight Scans is much faster, especially on larger maps with a low likelihood of cross-map visibility.



(a) Initial intervals (b) Resulting search tree

Figure 5: An All-Direction Line-of-Sight scan. The dark circle is the origin of the scan. Intervals are the horizontal lines. The found visible successors are also marked.

We initialise the scan around a point (the source) by generating horizontal intervals around it as shown in Figure 5a. Each interval is a tuple (x_L, x_R, y) consisting of an integer y-coordinate and two fractional endpoints on the x-axis. The successors of an interval are the observable successors defined in (Harabor and Grastien 2013), which are computed by projecting the current interval onto the next y-coordinate away from the source. Obstacles split up generated intervals. An example is shown in Figure 6.

From there, we conduct a depth-first search over the intervals, with each interval generating its observable successors, forming the search tree in Figure 5b. As visibility graph vertices only occur at the endpoints of the intervals, it suffices to check the interval endpoints to obtain the list of visible successors. We call this an All-Direction Line-of-Sight Scan.

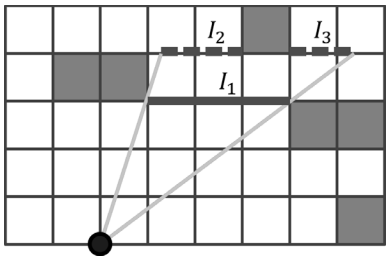


Figure 6: Successor intervals I_2 and I_3 generated from I_1.

In a Sparse Visibility Graph, we only add edges to vertices in taut regions (Figure 2a), which are determined by the current vertex's adjacent obstacles. Figure 7 illustrates the six different cases. Thus, for each vertex we need only scan within the taut regions. We do this by simply changing the initial states of the search as shown in Figure 8. We call this a Taut-Direction Line-of-Sight Scan.

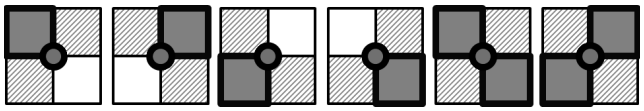
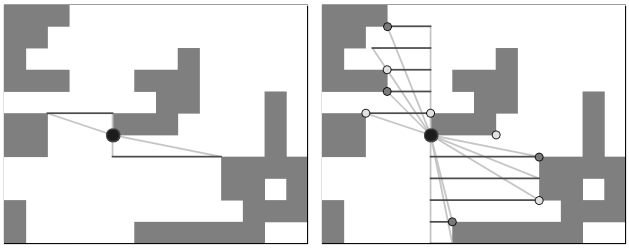


Figure 7: The six different obstacle (dark grey, outlined) configurations that determine taut regions (shaded).



(a) Initial intervals (b) Resulting search tree

Figure 8: A Taut-Direction Line-of-Sight scan. The found visible neighbours with a darker shade in (b) are also pruned as they do not meet the condition shown in Figure 3.

Line-of-Sight Scans can be sped up by pre-computing left and right extents for each grid vertex - the number of tiles one can traverse in that direction before hitting an obstacle, as shown in implementation of Anya in (Uras and Koenig 2015a). This pre-computation also improves the runtime speed of the algorithm, as All-Direction Line-of-Sight Scans are used to insert start and end points into the graph.

Properties of Sparse Visibility Graphs

The Sparse Visibility Graph Algorithm simply uses Taut A* over a pre-processed SVG. SVGs reduce the average vertex degree with no cost to optimality. On randomly generated maps with percentages of blocked tiles ranging between 6% and 40%, the average vertex degree of VGs remains approximately 2.5 times that of SVGs. We see this in Figure 9.

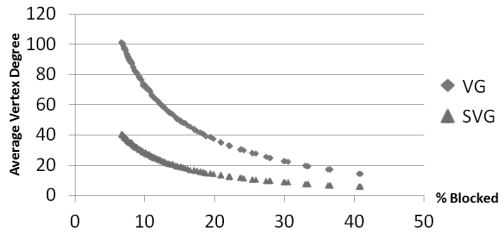
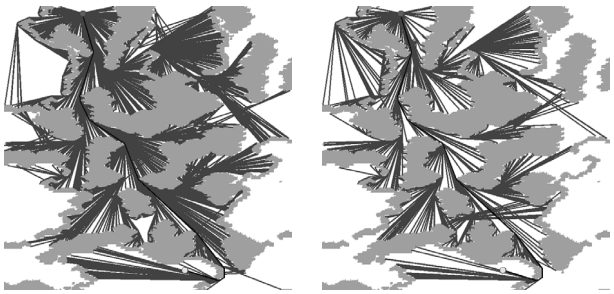


Figure 9: Comparison of average vertex degree on randomly-generated maps of various blocked densities.

Also, from Figure 10, we can see that the search tree of the Sparse Visibility Graph algorithm is more sparse than that of the original Visibility Graph algorithm.



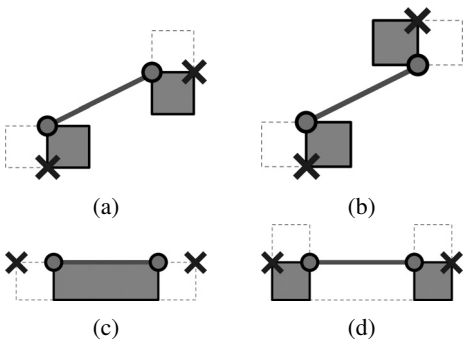
(a) VG (b) SVG

Figure 10: Search tree comparson on the map EbonLakes.

A key property of SVGs is that it almost cannot be pruned any further. Theorem 1 describes this property:

Theorem 1. *For each edge in the Sparse Visibility Graph, there exist two points which have an optimal path that uses that edge, neither of which are the endpoints of the edge.*

Proof. Edges in the SVG each belong to one of the four cases in Figure 11. In each case, the two required points are marked with crosses. The tiles that are necessarily unblocked in each case are marked with dotted lines. □



(a) (b) (c) (d)

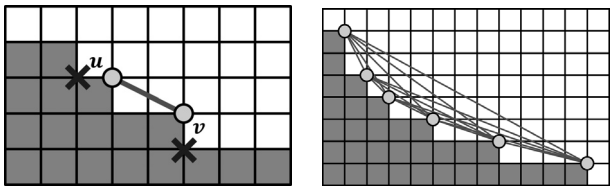
Figure 11: The four possible types of edge in SVGs.

We note that Theorem 1 does not guarantee that an edge is necessary in the case of multiple optimal paths between the two points. However, this is uncommon, so pruning these edges will yield only a marginal running time improvement.

Experimental evaluation of SVGs can be found in the Experiments section at the end of the paper.

Edge N-Level Sparse Visibility Graphs

Before we discuss ENLSVGs, it is important to understand the flaws of the SVG Algorithm. Even though Theorem 1 states that every edge in an SVG is necessary with a few rare exceptions, a large percentage of the edges are only useful for a small set of start-goal pairs. An example is Figure 12a, where edge uv is only useful for constructing a path between the two marked points. Figure 12b shows a clique of edges, each of which are useful for only a few start or goal points.



(a) (b)

Figure 12: Edges with limited usefulness within a concave section of blocked tiles.

Edge Levels

In SVGs, we prune outgoing edges that, when traversed, cannot be taut on the next hop. We extend this concept by looking further ahead than one hop, and prune outgoing edges that cannot be taut in future hops. We note that in SVGs, edges pruned due to a lack of taut exits are present in an optimal path only if they are the first or last hop of the search. We thus define the following concept of edge levels:

Definition 1. Edge Level
An edge is level $k \geq 0$ if at any one of its endpoints, it has no taut neighbouring edge of level more than $k-1$, and if $k > 0$, also has a taut neighbouring edge of level $k-1$. Edges that do not fit this definition have level ∞.

All "edges" not in the Sparse Visibility Graph are referred to as Level-0 edges. The idea is that for an edge e of level ℓ, for any taut path that passes through e, edge e must be the k^{th} hop from one of the endpoints of the path, for some $k \leq \ell$. If we were to restrict our search to only edges of increasing level from either end, all taut paths will be considered in the search, maintaining optimality.

These edge levels can be computed by iteratively pruning edges level-by-level as shown in Algorithm 1. Note that the computed edge levels are independent of the order the edges are selected in line 8. A simple algorithm is used for ease of understanding, though we believe that this procedure can be implemented with a more efficient algorithm.

Figure 13 illustrates an example of edge levels on an SVG. Edges of different levels are given different colour shades. Figure 14 highlights these edge levels more clearly.

Algorithm 1 ComputeEdgeLevels

```
1: procedure ComputeEdgeLevels(E)
2:     for each e = (u, v) ∈ E do
3:         e.level ← ∞
4:     hasChanges ← True
5:     ℓ ← 1
6:     while hasChanges do
7:         hasChanges ← False
8:         for each e = (u, v) ∈ E do
9:             if u or v has no taut exit of level ≥ ℓ then
10:                e.level ← ℓ
11:                hasChanges ← True
12:        ℓ ← ℓ + 1
```

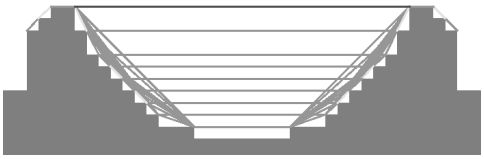


Figure 13: Edge levels in a Sparse Visibility Graph

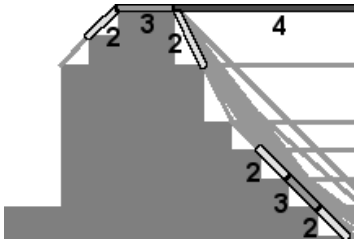


Figure 14: Zoom-in of Figure 13, with edges of level 2 and above outlined and labelled.

With the edge levels defined as before, we have the following results on the edge levels:

Lemma 2. *Consider any taut path. Let the levels of the edges along be the path be $\ell_1, \ell_2, \cdots, \ell_n$ respectively. Then for each $i \in \{2, 3, \cdots, n-1\}$, if ℓ_i is finite, then either $\ell_{i-1} < \ell_i$ or $\ell_{i+1} < \ell_i$.*

Proof. Suppose that there is an i such that $\ell_{i-1} \geq \ell_i$ and $\ell_{i+1} \geq \ell_i$. As the subpaths $e_{i-1}e_i$ and e_ie_{i+1} are taut, this means edge e_i has neighbouring edges on both endpoints with level $\geq \ell_i$, implying the level of e_i is at least $\ell_i + 1$, which is a contradiction. □

Theorem 3. *Assuming that every edge has a finite level, the sequence of edges of any taut path between the start and the goal vertices will be of the form*

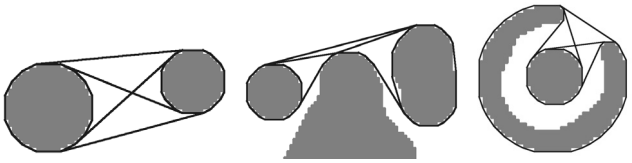
$$e_1 e_2 \cdots e_k e'_{k+1} \cdots e'_n$$

where edges $e_1e_2 \cdots e_k$ have strictly increasing levels, and $e'_{k+1}e'_{k+2} \cdots e_n$ have strictly decreasing levels.

Proof. Let the levels of the edges along be the path be $\ell_1, \ell_2, \cdots, \ell_n$ respectively. From the lemma, we can see that in the path, if there is an i such that $\ell_i \geq \ell_{i+1}$, then we must have $\ell_{i+1} > \ell_{i+2}$, implying $\ell_{i+2} > \ell_{i+3}$ and so on, inductively proving that the remaining edges of the path will have strictly decreasing levels, proving Theorem 3. □

Level-W Edges

Not all edges will be assigned a finite level by Algorithm 1. We refer to the remaining edges (with level ∞) as level-W edges. Notably, an edge is level-W if and only if it is part of some taut cycle. Examples of taut cycles are shown in Figure 15. We observe that the graph of level-W edges is similar to the tangent graphs described in (Liu and Arimoto 1992), as taut cycles wrap around the convex hulls of obstacles.



(a) (b) (c)

Figure 15: Examples of cycles of taut edges.

With this definition of level-W edges, we have a similar theorem regarding the edge levels of taut paths.

Theorem 4. *The sequence of edges of any taut path between the start and goal vertices will be of the form:*

$$e_1 e_2 \cdots e_{k_1} w_{k_1+1} w_{k_1+2} \cdots w_{k_2} e'_{k_2+1} e'_{k_2+2} \cdots e'_n$$

where edges $e_1e_2 \cdots e_{k_1}$ have strictly increasing levels, $w_{k_1+1}w_{k_1+2} \cdots w_{k_2}$ are level-W, and $e'_{k_2+1}e'_{k_2+2} \cdots e_n$ have strictly decreasing levels.

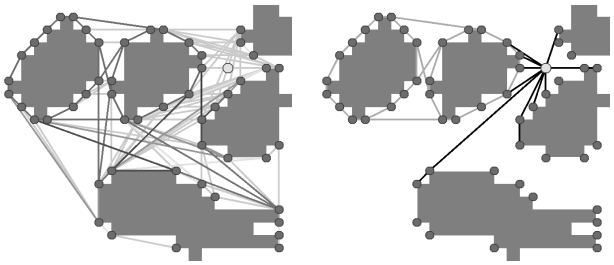
Proof. The proof is similar to that of Theorem 3. Let the levels of the edges along be the path be $\ell_1, \ell_2, \cdots, \ell_n$ respectively. From Lemma 2, we can see that in the path, if there is an i such that $\ell_i \geq \ell_{i+1}$ and ℓ_{i+1} is finite, then we must have $\ell_{i+1} > \ell_{i+2}$, implying $\ell_{i+2} > \ell_{i+3}$ and so on, inductively proving that the remaining edges of the path will have strictly decreasing levels. This gives the required form. □

As the optimal path is taut, it obeys the rule in Theorem 4. Thus, other than near the start and the end of the route, we only have to search level-W edges to ensure that the optimal path is included in the search.

Edge Marking and Search

Edge Marking is how we make use of Theorem 4 in the search. Before the search, from both the start and goal vertices, we run a depth-first search to mark all finite-level edges reachable by a taut path of strictly increasing edge levels. We stop the search when we reach Level-W edges. Figure 16 illustrates the edges that are marked this way.

We then restrict our A* search to use only marked edges and level-W edges. Let H denote the subgraph induced by



(a) Before Marking (b) After Marking

Figure 16: Illustration of the edge marking process.

the marked and level-W edges. To prove that the algorithm is optimal, it suffices to show that the optimal path resides within the graph H.

Theorem 5. *All taut paths (including the optimal path) from the start s to the goal t reside within the graph H.*

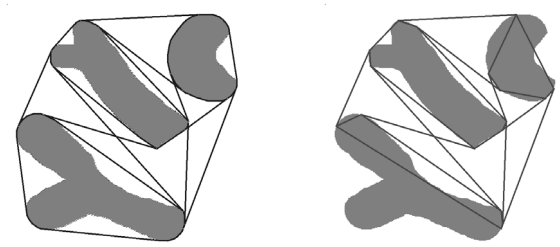
Proof. Consider any taut path from s to t. It must have edges of the form in Theorem 4. Edges $e_1, e_2 \cdots, e_{k_1}$ would have been marked from s as each of these edges are reachable by a taut path of increasing edge levels from s. Similarly, edges $e'_{k_2+1}, \cdots e'_n$ will be marked from t. Thus all of the edges in the path are either marked or Level-W, and so are in H. □

Thus the algorithm can be summarised in three steps:

1. Insert start, goal into the graph using Line-of-Sight Scans.
2. Mark reachable edges from the start and goal vertices.
3. Compute optimal path to the goal by running Taut A* on only the marked and Level-W edges.

Skip-Edges

The graph of level-W edges can be further reduced through the concept of Skip-Edges. As seen in Figure 17a, large convex hulls can produce long, unbranching paths of level-W edges. Each unbranching path can be reduced to a single edge with weight equal to the length of the path as shown in Figure 17b. We refer to these edges as Skip-Edges. Skip-Edges makes the search time dependent on the amount of detail in the map, rather than the scale of the map.

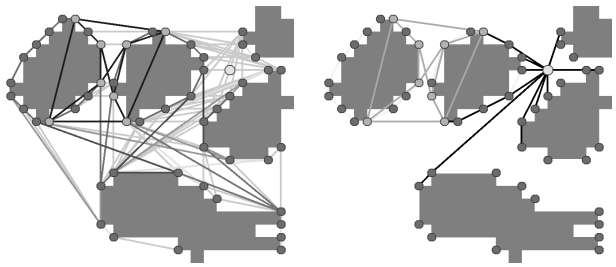


(a) level-W Edges (b) Skip-Edges

Figure 17: Skip-Edge network derived from level-W edges.

To construct the Skip-Edge network, consider the graph W induced by the level-W edges. All vertices of degree at least 3 in W are identified as Skip-Vertices. We then trace the unbranching paths of Level-W edges between Skip-Vertices to form the Skip-Edge network.

We also make a slight change to the marking scheme. If we reach a Level-W edge while marking edges of increasing level, we continue marking subsequent Level-W edges until a Skip-Vertex is reached. This is illustrated in Figure 18. We then run Taut A* on only marked edges and Skip-Edges.

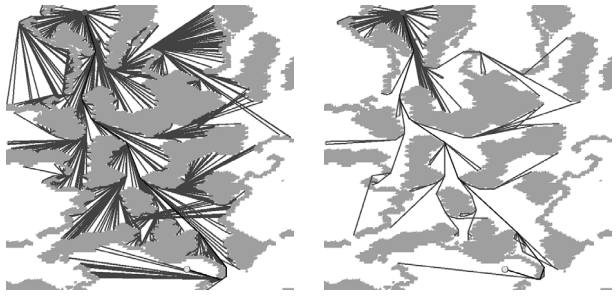


(a) Before Marking (b) After Marking

Figure 18: The edge marking process with Skip-Edges.

Search Tree Comparison

Figure 19 illustrates the difference between the search trees of the SVG and ENLSVG algorithms. We can observe the running time improvement through how much of the original search tree the algorithms prune.



(a) SVG (b) ENLSVG

Figure 19: Search tree comparson on the map EbonLakes.

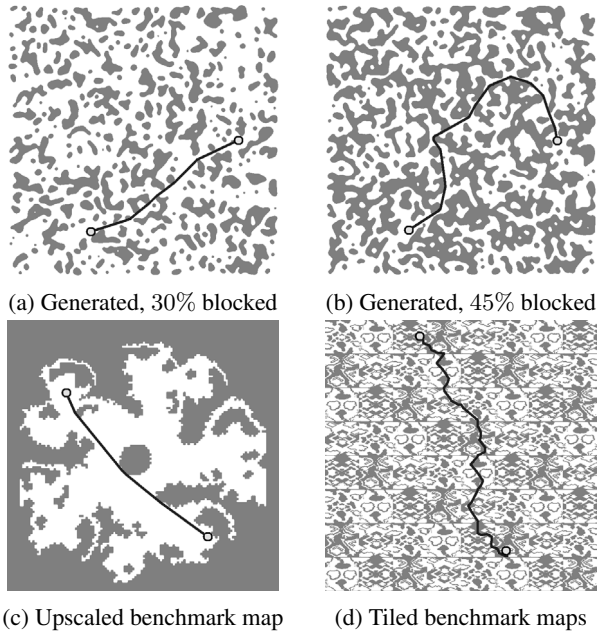
Experimental Results

Experimental Setup

The algorithms compared are Theta*, Anya (Anya16), the algorithm of Shah and Gupta (SG16), the Visibility Graph Algorithm, SVGs and ENLSVGs. For the Visibility Graph Algorithm, we implement it using Line-of-Sight Checks (VG_C), Rotational Plane Sweeps (VG_{RPS}) and Line-of-Sight Scans (VG_S). The relationship between Theta* and other algorithms can be found in (Uras and Koenig 2015a).

Other than Anya16, we have implemented all the algorithms listed above in Java[1]. Anya16 refers to the recent implementation of Anya in (Harabor et al. 2016), also in Java.

[1]The implementations are available at github.com/Ohohcakester/Any-Angle-Pathfinding



(a) Generated, 30% blocked (b) Generated, 45% blocked

(c) Upscaled benchmark map (d) Tiled benchmark maps

Figure 20: Some 4000×4000 maps used in the experiments.

Note that this implementation contains multiple optimisations that makes it perform much better than the implementation given in (Uras and Koenig 2015a).

In addition to the 512×512 benchmarks from (Sturtevant 2012), we use three methods to generate larger maps. The first two sets are generated using cellular automata (Johnson, Yannakakis, and Togelius 2010), a common technique for generating cave-like game maps, with 30% / 45% blocked tiles respectively. The third set is upscaled versions of benchmark game maps, smoothed using cellular automata. The fourth set is generated by tiling benchmark game maps to form larger maps. Examples are shown in Figure 20. Map sizes used are around 2000×2000, 4000×4000, and 6000×6000. Running times on each map are averaged over 1000 to 2500 runs using 100 randomly-picked pairs of reachable points. All times are given in milliseconds. All experiments were run on a 3.40 GHz Intel Core i7-6700 CPU with 8GB RAM.

Comparison of Construction Times

As the Visibility Graph algorithms are offline algorithms, we also measure the time taken to preprocess the Visibility Graph. We compare the construction times of the Visibility Graph variants VG_C, VG_{RPS}, VG_S, SVGs and ENLSVGs. SVGs and ENLSVGs use Line-of-Sight Scans. The construction times are given in Table 3.

Rotational Plane Sweeps (VG_{RPS}) runs the slowest out of the three variants, by a large margin. The improvement in construction time from using Line-of-Sight Scans (VG_S) over the other two methods is especially significant on random maps. SVG construction is faster than VG_S as it takes advantage of Taut-Direction Line-of-Sight Scans. ENLSVGs includes an additional hierarchy-building step after the construction of the SVG, and thus is slightly slower.

Comparison of Line-of-Sight Algorithms

The choice of Line-of-Sight algorithm also affects the time taken to insert the start and goal points into a Visibility Graph during a query. A large proportion of the time saving from VG_C to SVGs comes from using Line-of-Sight Scans over Line-of-Sight Checks for insertion. This is reflected in the running time of VG_S in Table 1. As Rotational Plane Sweeps have been shown to be much slower than Line-of-Sight Checks, we omit it from our running time tests.

This is because the running time of Line-of-Sight Checks depends on the number of visibility graph vertices (convex corners) in the entire grid, while the running time of Line-of-Sight Scans depends on the size of the visible region from each vertex. Random maps have more convex corners and smaller visible regions. The Rotational Plane Sweep Algorithm is implemented by tracing the boundaries of each obstacle to form polygons. This makes the algorithm very slow as these polygons have a large number of edges.

Running Time Breakdown

We break down the ENLSVG algorithm into three components: insertion, marking and search. Insertion is the Line-of-Sight Scans to connect the start and goal to the graph. Search refers to the final A* search. Only the insertion and search components apply to the SVG algorithm. The breakdown of the running times is given in Table 2.

We see that ENLSVGs perform a lot better than SVGs on tiled and generated maps, but only slightly better on upscaled maps. This is because the bottleneck on upscaled maps is the insertion and marking steps, while the bottleneck on tiled maps is the search step. Insertion time depends on how wide the open spaces are, while search time is tied to the complexity of the map. Upscaled maps have large open spaces and low complexity, while tiled maps are the opposite. If we look at search time alone however, we see that ENLSVGs consistently do much better than SVGs.

Comparison with other Algorithms

As SVGs are simply less dense Visibility Graphs, SVGs improve the running time from VG_S with no additional cost. While SVGs effectively cut running time regardless of map structure, ENLSVGs speed up search by taking advantage of map structure to build a hierarchy. As such, the cost savings are small on completely random maps (Table 1).

When compared to existing algorithms VG_C, Theta*, Anya16 and SG16, especially on large maps, SVGs and ENLSVGs are at least an order of magnitude faster. We find that Anya16 also performs well even on large maps, while SG16 and Theta* degrades quickly on certain maps.

In SG16, the complexity of the map is indicated by the number of quadtree nodes representing it. Thus, for a better comparison, we have included the number of quadtree nodes required to represent each the maps (on average) in Table 1.

SG16 improves on the naive Rotational Plane Sweep algorithm by dividing the map into nonintersecting convex hulls. From our experiments, we find that SG16 performs

Map Set	Size	Quadtree Nodes	Theta*	SG16	Anya16	VG_C	VG_S	SVG	ENLSVG
bg512	512^2	9443.2	3.01	12.39	0.12	0.21	0.23	0.04	0.05
sc1	512^2	30209.2	22.20	22.34	1.10	2.74	1.06	0.39	0.19
wc3	512^2	13722.3	3.12	12.18	0.16	0.38	0.28	0.05	0.07
random10	512^2	123904.6	2.82	-	2.96	8.17	4.22	0.86	1.00
random20	512^2	179588.2	4.44	-	10.97	11.41	5.37	1.82	1.74
random30	512^2	209655.4	5.73	-	16.23	10.90	5.37	2.45	2.26
random40	512^2	145179.1	6.01	-	13.83	6.27	3.74	2.01	1.58
gen30_2000	2000^2	214479.0	101.22	145.43	2.99	18.37	4.71	1.30	0.98
gen30_4000	4000^2	867156.0	421.03	1517.17	12.72	76.72	16.40	5.16	1.73
gen30_6000	6000^2	1953102.0	991.00	8158.56	32.55	168.89	34.63	12.08	3.16
gen45_2000	2000^2	242387.0	185.09	316.55	6.53	17.92	7.16	2.51	0.63
gen45_4000	4000^2	976729.0	832.99	2454.35	23.80	75.73	27.61	10.33	1.18
gen45_6000	6000^2	2192387.0	2412.02	10882.95	63.80	177.11	73.97	28.16	1.80
scaled_2048	2048^2	58070.5	225.67	157.12	1.10	10.14	2.50	0.56	0.43
scaled_4096	4096^2	106296.5	1429.99	698.81	2.39	47.66	8.07	1.76	1.51
scaled_6144	6144^2	180578.0	4665.53	1624.11	4.05	102.75	16.25	3.56	3.29
tiled_2048	2048^2	256104.3	163.31	338.50	6.50	14.24	4.31	1.77	0.29
tiled_4096	4096^2	1041379.0	751.63	3133.74	24.61	69.75	18.76	8.81	0.74
tiled_6144	6144^2	2353286.5	1911.95	13665.52	57.04	145.34	46.67	23.26	1.87

Table 1: Running times (in milliseconds) for benchmark maps, as well as generated, scaled and tiled maps.

	SVG		ENLSVG		
Map Set	Insert	Search	Insert	Marking	Search
bg512	0.0214	0.0212	0.0222	0.0261	0.0053
sc1	0.0317	0.3616	0.0333	0.1199	0.0372
wc3	0.0236	0.0236	0.0248	0.0374	0.0114
random10	0.0218	0.8381	0.0280	0.1584	0.8098
random20	0.0081	1.8114	0.0099	0.0410	1.6888
random30	0.0046	2.4404	0.0057	0.0176	2.2403
random40	0.0026	2.0078	0.0032	0.0083	1.5699
gen30_2000	0.2100	1.0926	0.2230	0.6372	0.1214
gen30_4000	0.3229	4.8377	0.3278	0.8996	0.5037
gen30_6000	0.4052	11.6717	0.5942	1.1845	1.3771
gen45_2000	0.1224	2.3805	0.1291	0.3337	0.1680
gen45_4000	0.1974	10.1294	0.1790	0.4398	0.5606
gen45_6000	0.2236	27.9335	0.1908	0.4715	1.1323
scaled_2048	0.1151	0.4424	0.1159	0.2900	0.0277
scaled_4096	0.3002	1.4570	0.3648	1.0894	0.0508
scaled_6144	0.5852	2.9704	0.8392	2.3696	0.0803
tiled_2048	0.0427	1.7245	0.0395	0.1052	0.1450
tiled_4096	0.0746	8.7308	0.0438	0.1184	0.5735
tiled_6144	0.0944	23.1664	0.0743	0.1469	1.6480

Table 2: Running time (ms) breakdown of the ENLSVG and SVG algorithms.

well on large maps with few details. However, performance degrades quickly with the amount of detail in the map. As compared to Theta*, SG16 performs better on large maps with few details, while Theta* performs relatively better on smaller maps with more details. SG16 was unable to run some of the benchmarks within a reasonable amount of time. For the rest of the benchmarks, we ran SG16 with only one trial per test case, rather than the usual 25.

ENLSVGs perform well (in real-time, on the order of milliseconds per computation) even on 10000×10000 grids. Memory constraints from storing large grids however prevent us from extracting reliable running time data, due to inconsistent running times when memory paging occurs.

Map Set	VG_C	VG_{RPS}	VG_S	SVG	ENLSVG
bg512	37	398	9	7	10
sc1	1688	20070	58	47	90
wc3	81	667	15	11	14
random10	81847	1592350	1834	378	1376
random20	132736	4364323	728	206	636
random30	130427	5708782	378	127	376
random40	30445	1864484	126	51	139

Table 3: Graph construction times (ms) on benchmark maps.

Conclusions

On maps with wider open spaces, even though the ENLSVG algorithm's running time is bottlenecked by the insertion and marking steps, the ultimate reduction in time used for the search step gives is an indication of the potential speedup that can be obtained through the use of ENLSVGs.

If we could do away with the insertion and marking steps, this speedup could be achieved. For example, pre-processing could be used to quickly find the visible neighbours of the start and goal points. Regarding the marking step, we can see that some form of goal-based initial search is needed in this algorithm, in order for the search to "know" that it is approaching the goal, and start following paths of decreasing edge levels. Whether a double-ended search algorithm can be used to omit the marking step is an open question.

In this paper, we also see the use of taut (locally optimal) paths as a heuristic for globally optimal paths to reduce the search space. The exact relationship between ENLSVGs (edge levels by pruning non-taut paths) and the 8-directional optimal algorithm N-Level Subgoal Graphs (vertex levels by pruning suboptimal paths), as well as a deeper analysis of their running times and pitfalls, remains to be investigated.

References

Algfoor, Z. A.; Sunar, M. S.; and Kolivand, H. 2015. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*.

Bresenham, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems journal* 4(1):25–30.

Choset, H. M. 2005. *Principles of robot motion: theory, algorithms, and implementation*. MIT press.

Harabor, D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In *AAAI*.

Harabor, D., and Grastien, A. 2013. An optimal any-angle pathfinding algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 308–311.

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

Johnson, L.; Yannakakis, G. N.; and Togelius, J. 2010. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 10. ACM.

Liu, Y.-H., and Arimoto, S. 1992. Path planning using a tangent graph for mobile robots among polygonal and curved obstacles communication. *The International Journal of Robotics Research* 11(4):376–382.

Lozano-Prez, T., and Wesley, M. A. 1979. An algorithm for planning collison-free paths among polyhedral obstacles. *Communications of the ACM* 22:560–570.

Nash, A.; Daniel, K.; Koenig, S.; and Felner, A. 2007. Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1117–1183.

Oh, S., and Leong, H. W. 2016. Strict theta*: Shorter motion path planning using taut paths. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Shah, B. C., and Gupta, S. K. 2016. Speeding up a* search on visibility graphs defined over quadtrees to enable long distance path planning for unmanned surface vehicles. In *Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling*, 527–535. AAAI Press.

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. 2015a. An empirical comparison of any-angle path-planning algorithms. In *Proceedings of the Annual Symposium on Combinatorial Search*.

Uras, T., and Koenig, S. 2015b. Speeding-up any-angle path-planning on grids. In *ICAPS*, 234–238.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *ICAPS*.

Yap, P.; Burch, N.; Holte, R.; and Schaeffer, J. 2011. Block A*: Database-driven search with applications in any-angle path-planning. In *AAAI*.