Minimal Construct: Efficient Shortest Path Finding for Mobile Robots in Polygonal Maps

Marcell Missura[1], Daniel D. Lee[2], and Maren Bennewitz[1]

***Abstract*— With the advent of polygonal maps finding their way into the navigational software of mobile robots, the Visibility Graph can be used to search for the shortest collision-free path. The nature of the Visibility Graph-based shortest path algorithms is such that first the entire graph is computed in a relatively time-consuming manner. Then, the graph can be searched efficiently any number of times for varying start and target state combinations with the A* or the Dijkstra algorithm. However, real-world environments are typically too dynamic for a map to remain valid for a long time. With the goal of obtaining the shortest path quickly in an ever changing environment, we introduce a rapid path finding algorithm—Minimal Construct—that discovers only a necessary portion of the Visibility Graph around the obstacles that actually get in the way. Collision tests are computed only for lines that seem heuristically promising. This way, shortest paths can be found much faster than with a state-of-the-art Visibility Graph algorithm and as our experiments show, even grid-based A* searches are outperformed in most cases with the added benefit of smoother and shorter paths.**

I. Introduction

The research of mobile robots has experienced significant breakthroughs in the last decades. Simultaneous localization and mapping (SLAM) and the navigation of ground vehicles have matured to a state where robots are able to move about in realistic office environments for days [1] without human intervention. Software components that accomplish this task are widely available within the ROS [2] framework. The underlying philosophy of the state-of-the-art navigation software hinges on a rectangular grid-based occupancy map where each cell of the grid represents a small spatial unit of the environment that can be blocked, free, or unknown. This map can then be searched, cell by cell, for a shortest path.

Polygonal maps are a promising alternative to grids due to a number of advantages that can be leveraged to outperform the success of the cell decomposition approach. Geometric arrangements have a more compact memory footprint and are also well suited to represent moving obstacles. They are not prone to the pitfalls of discretization. Most importantly for our focus of finding shortest paths, a polygonal map gives rise to the Visibility Graph. The Visibility Graph consists of the edges of the polygons in the scene and additional edges that connect the pairs of polygon corners that can "see" each other, i. e., the line segment connecting the corners does not intersect any of the polygons. It is well known that shortest paths can be found in this graph [3]. Once the graph has been constructed, smooth paths of optimal length and a minimal number of line segments can be found quickly with the A* algorithm as opposed to the jagged and suboptimal paths emerging from a grid. An illustration is shown in Figure 1.

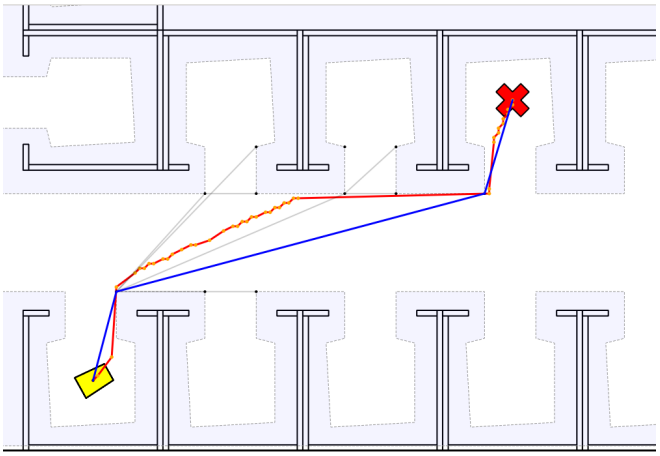


Fig. 1: Obstacle avoiding shortest paths in a polygonal map. The walls were expanded by the size of the robot to polygonal areas shown in light blue. The graph explored by our Minimal Construct algorithm (shown with thin grey lines) is only a small fraction of the entire Visibility Graph of this map. While the path found by an A* search (shown in red) in an equivalent grid is a jagged path of 38 pieces and suboptimal length, the result of our Visibility Graph-based Minimal Construct algorithm (blue) is optimal in length and has a minimal complexity of three line segments.

The construction of the Visibility Graph, however, requires much more computation time than the search. The best known algorithms can compute the Visibility Graph using $O(n^2)$ operations where n is the number of the polygon edges. Consequently, the construction time of the graph is sensitive to the complexity of the map. At the same time, frequent changes in the environment such as moving people and objects, opening and closing doors, or simply the exploration of an unknown part of the map, invalidate the graph. It has to be repaired or reconstructed, which slows down the process of finding the shortest path.

The Minimal Construct algorithm presented in this work mitigates this issue by computing only the necessary portion of the Visibility Graph during an A* search rather than before. Graph edges are only checked for intersection with polygons when they are popped from the priority queue. Hence, the most expensive part of the computation—a large number of line intersection tests—is reduced to those lines that seem heuristically promising. The search always begins with the minimal graph consisting of a straight line from the

[1]Marcell Missura and Maren Bennewitz are with Humanoid Robots Lab, University of Bonn, Germany

[2]Daniel D. Lee is with the GRASP Laboratory, University of Pennsylvania, USA

start to the target. If the line is blocked by an obstacle, the corners of the obstacle are connected to the graph and the A* search is continued in this updated graph until the target is reached, or, in the worst case, the entire Visibility Graph is discovered.

Further below, we present experiments where we systematically increase the complexity of a map and show that the Minimal Construct algorithm consistently outperforms one of the fastest known $O(n^2)$ algorithms [4] in terms of finding the shortest path while having the same asymptotic complexity. Leveraging the lower computational cost, shortest paths in polygonal maps can be recomputed more frequently to account for changes in the environment. We also include two grid-based alternatives in our experiments, a standard eight-connected A* search and the Lazy Theta* algorithm, for the comparison of runtimes that can be achieved by searching a grid.

II. Related Work

The earliest works involving mobile navigation using polygonal maps date back to the experiments performed with the SHAKEY robot [5]. Even if the first layer of perception has been a grid, the corners of grid segments were used for navigation in a Visibility Graph-like manner.

Lozano-Perez et al. [3] were the first to propose using shortest paths in the Visibility Graph for robot motion planning. Their concept included the growing of the polygons by the radius of the robot such that the robot can be regarded as a point.

Lee [6] introduced a more efficient algorithm to compute the Visibility Graph in $O(n^2 \log n)$ time by using a radial sweep technique. Asano et al. [7] proposed two $O(n^2)$ algorithms, one based on triangulation and the other on a polar sweep performed in the dual arrangement one vertex at a time. A key feature of the sweep is a table lookup that allows to find and remove a segment from a sorted list in constant rather than in $O(\log n)$ time. Welzl [8] also proposed an $O(n^2)$ algorithm based on a polar sweep that is performed for all vertices simultaneously. The author exploits the fact that a partial order over the slope of the line segments is sufficient for the sweep to compute the correct graph and a full sorting in $O(n^2 \log n)$ can be avoided. The partial order is computed in the dual arrangement with a topological sort. Later on, Overmars and Welzl [4] presented a simpler version of their algorithm using a rotation tree instead of the dual arrangement. This version also needs less space in memory. Ghosh and Mount [9] developed an output sensitive algorithm using a triangulation and funnel splits in a planar sweep that computes in $O(e+n \log n)$ where e is the number of the edges in the output. The algorithm is optimal for sparse graphs. However, the implementation of this algorithm is more difficult than the others and it does not always perform better than the $O(n^2)$ variants [10]. We are using an implementation of the Overmars Welzl [4] algorithm to evaluate the performance of our method.

Rohnert [11] devised an $O(n + f^2 \log n)$ algorithm where $f < n$ is the number of polygons in the scene. This is not necessarily faster, but the work shows that between two convex and disjoint polygons only the polygon edges and up to four tangential lines can possibly contribute to the shortest path. Unfortunately, convexity cannot be assumed in realistic environments, but we inherit the use of tangential edges for non-convex obstacles and can significantly reduce the number of edges in the graph this way.

Huang at al. [12] approached the shortest path problem similar to us in a way that they first identify a portion of the Visibility Graph that the shortest path resides in. A rectangular shaped active region around the direct line between start and target is grown by adding more and more obstacles to it until it is certain that the shortest path is contained in the box. Then, the Visibility Graph is computed using only the vertices that are inside the active region. However, this approach has only been evaluated on a few small examples and the reported runtimes are orders of magnitudes larger than ours.

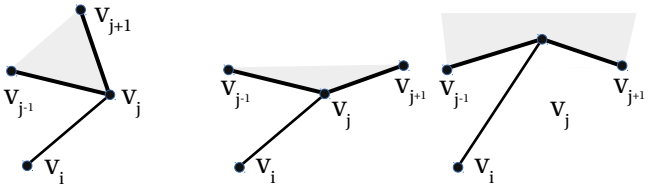
The A* algorithm [13] has been established as the de facto standard for grid searching. Anytime versions [14] have been investigated that inflate the heuristic to gain a suboptimal solution in shorter time, and improve it with subsequent searches until deliberation time runs out. Dynamic variants such as D* Lite [15] are designed to be used by robots in motion that discover unexpected changes in the environment and keep previously computed information in order to speed up replanning. Any-angle variants like Field-D* [16] and Incremental Phi* [17] produce smoother paths using interpolation between actions or connecting a straight line back to earlier nodes in the path rather than to the direct parent. Lazy Theta* [18] backtracks an open node to its parent by performing a line of sight check in the grid only when the node is popped from the queue and is similar in spirit to our algorithm. It produces smoother and shorter paths than A* at the expense of the additional line-of-sight tests.

In addition to the Overmars Welzl Visibility Graph construction method, we chose a standard eight-connected A* as it was used in [1] as a benchmark of computational performance, and we favor the Lazy Theta* [18] algorithm due to its superior simplicity and efficiency as an any-angle A* search to represent an example that produces high-quality paths in a grid.

III. Minimal Construct

A. Preliminaries

The most simple way of constructing the Visibility Graph is an $O(n^3)$ algorithm that enumerates all $O(n^2)$ pairs of vertices and tests the edges between them for intersection with each of the n lines in the scene. This algorithm involves the computation of a large number of line intersections. More sophisticated algorithms, e. g., our favored Overmars Welzl rotation tree [4], sweep the visible contour around a point and keep track of the nearest visible line. During this polar sweep, edges appear as events sorted by angle. Each event reports visibility between two vertices and may update the nearest visible line. Since there are $O(n^2)$ edges and every edge is handled in amortized constant time, the entire algorithm



(a) Tangential edge (b) Non-tangential edge (c) Concave corner

Fig. 2: a) An edge (v_i, v_j) is tangential in point v_j if the adjacent corners of the polygon v_{j-1} and v_{j+1} lie on the same side of the edge. If the edge (v_i, v_j) is non-tangential b), or the corner v_j is concave c), the entire area of the triangle $\Delta\,(v_i, v_{j-1}, v_{j+1})$ can be seen from v_i and (v_i, v_j) cannot be a part of a shortest path.

finishes after no more than $O(n^2)$ operations. However, a change of the graph later on cannot be accommodated easily, as for example the rotation tree is consumed during the sweep and needs to be rebuilt.

Minimal Construct does not attempt to compute the entire Visibility Graph. Typically, only a small portion of the graph is discovered during the search for the shortest path. In order to avoid collision checking of line segments that do not directly contribute to the shortest path, the computation of line intersection tests is delayed until they become necessary. The line intersection tests that the algorithm does perform are accelerated with the help of bounding boxes. When the bounding box of an edge does not intersect the bounding box of a polygon, none of the edges of that polygon need to be tested for line intersection. Bounding box tests are very fast to compute and increase the overall performance of the algorithm. Furthermore, we exploit the fact that concave corners and non-tangential edges can never be a part of a shortest path. An edge (v_i, v_j) is tangential in a vertex v_j if both polygon corners v_{j-1} and v_{j+1} adjacent to v_j lie on the same side of (v_i, v_j). An example is shown in Figure 2. Concave corners and edges that are not tangential in both ends are discarded as soon as they are discovered.

B. The Minimal Construct Algorithm

We assume a geometric scene of disjunct and non-convex polygons $\boldsymbol{S}$ consisting of n lines that intersect only in their end points.[1] The graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ that Minimal Construct discovers during the search contains a subset of the end points of the lines, i. e., $\boldsymbol{V} = \{v_{i<n}\}$ is a subset of the corners of the polygons, and edges $\boldsymbol{E} = \{(v_i, v_j)\,|v_i, v_j \in \boldsymbol{V}, i \neq j\}$ connecting pairs of these vertices. The start state s and the goal state t are treated as two additional vertices in the graph.

For the operation of the A* algorithm, which the Minimal Construct Algorithm is based on, we need to define a unidirectional parent relationship that is used to extract the shortest path by following the path from the target parent to parent to the start after the search has terminated. The functions PARENTOF(v_i), SETPARENT(v_i, v_j), and REMOVEPARENT(v_i) used by Algorithm 1 work as one would expect with SETPARENT(v_i, v_j) setting vertex v_i as parent of v_j and REMOVEPARENT(v_i) canceling the parent of v_i. A* opens vertices when they are pushed into the priority queue and closes them after they have been popped from the queue. For this, we use the functions ISOPEN(v_i), CLOSE(v_i), and ISCLOSED(v_i) to query and manipulate the open and closed state of vertices.

Algorithm 1 shows the pseudo code of our algorithm, which we describe in the following in detail. Minimal Construct initializes the search with the trivial one-edge graph from start to target by connecting s and t as neighbors, setting s as parent of t, and pushing the *target* vertex into the priority queue (lines 1 to 7 in Alg. 1). Note that s is initialized to be closed and remains closed at all times. Then, the program enters a loop where vertices are subsequently popped from the queue. Each time a vertex v is popped, the edge (u, v) between the vertex v and its parent u is tested for line intersection with the scene (line 11). This is a rather expensive $O(n)$ operation.

If the edge (u, v) is found to be collision-free in line 12, Minimal Construct can perform an A* expansion step in lines 13 through 28. This involves checking if the target has been reached, otherwise closing the vertex v and expanding all its neighbors. The expansion step applies a closed and open check in lines 19 and 20. v is set as parent of neighbor v_i, g, h, and f are computed, and v_i is pushed into the queue.

If the edge (u, v) is blocked, a new polygon $\boldsymbol{p}$ has been discovered. The blocked edge is removed from the graph in line 30. Then, vertex v is orphaned and reparented to the vertex in the graph with the lowest path cost g that is closed and has v as a neighbor. If such a parent has been found, the path cost $g(v)$ and the priority $f(v)$ are updated and v is pushed back into the queue. The parenting procedure is shown in Algorithm 2. By reparenting v and pushing it back into the open list, we maintain the completeness of the A* search by setting it into the state it would have had if the invalid edge (u, v) had not been present in the graph. Note that v may remain entirely without a parent until it is perhaps touched again at a later time.

After the reparenting in lines 31 and 32 of Algorithm 1, the polygon $\boldsymbol{p}$ that blocked the edge (u, v) is added to the graph in line 34 by calling Algorithm 3. The obstacle is connected by adding each of its convex corners to the graph and connecting them as a neighbor of every known vertex in the graph so far, if the edge is tangential in both ends. Following the linking into the graph, Algorithm 3 attempts to find a parent for each polygon corner by calling Algorithm 2. This preserves completeness by accommodating the fact that if the A* search had known about the edges to the new polygon, any of the new corners may have been pushed into the queue at an earlier time. Polygons are closed after discovery in line 35 of Algorithm 1 to make sure they are added to the graph only once. The procedure is repeated until either a path is found or the entire graph has been explored and it is certain that no solution exists.

[1]Note that the Minimal Construct algorithm works also in a scene of intersecting, non-convex polygons. We increased our input requirements in order to stay comparable with the Overmars Welzl algorithm.

Algorithm 1 Minimal Construct

Input: Set of polygons $\boldsymbol{S}$, Start vertex s, Target vertex t
Output: Path $\boldsymbol{P}$

```
1: Priority q                                   ▷ We are using a priority queue q
2: Graph G = (V, E)                             ▷ Start with an empty Graph
3: V ← V ∪ {s, t}                               ▷ Add start and target vertices
4: E ← E ∪ (s, t)                               ▷ Add t to the neighbors of s
5: SETPARENT(s, t)                              ▷ Set s as the parent of t
6: CLOSE(s)                                     ▷ Close the start vertex s
7: PUSH(t, q)                                   ▷ Push the target into the priority queue
8: while (q is not empty) do
9:    v ← POP(q)                                ▷ Pop the vertex v with the lowest f
10:   u ← PARENTOF(v)                           ▷ Get the parent of v
11:   Polygon p ← LINEINTERSECTIONTEST(S, (u, v))
12:   if (p == nil) then                        ▷ If no polygon has been intersected
13:      if (v = t) then                        ▷ If the target has been reached
14:         P ← EXTRACTPATH(v)                  ▷ Follow parents to start
15:         return P                            ▷ Finished!
16:      end if
17:      CLOSE(v)                               ▷ Close the vertex v
18:      for all (v_i, (v_i, v) ∈ E) do         ▷ Expand neighbors of v
19:         if (!ISCLOSED(v_i)) then            ▷ If v_i is not closed yet
20:            if (!ISOPEN(v_i) or g(v) + |v − v_i| < g(v_i)) then
21:               SETPARENT(v, v_i)             ▷ Set v as parent of v_i
22:               g(v_i) ← g(v) + |v − v_i|     ▷ Path cost g so far
23:               h(v_i) ← |v_i − t|            ▷ Heuristic h to target
24:               f(v_i) ← g(v_i) + h(v_i)      ▷ Priority f
25:               PUSH(v_i, q)                  ▷ Push v_i into the priority queue
26:            end if
27:         end if
28:      end for
29:   else                                      ▷ A polygon p has been intersected
30:      E ← E \ (v, u)                         ▷ v is no longer neighbor of u
31:      REMOVEPARENT(v)                        ▷ Remove parent of v
32:      FINDPARENT(v)                          ▷ (Algorithm 2)
33:      if (!ISCLOSED(p)) then                 ▷ If polygon p is not closed
34:         CONNECTOBSTACLE(p)                  ▷ (Algorithm 3)
35:         CLOSE(p)                            ▷ Close the intersected polygon p
36:      end if
37:   end if
38: end while
39: return P                                    ▷ Search failed. Return empty Path
```

Algorithm 2 Find Parent

Input: Vertex v

```
1: minPathCost ← inf          ▷ Init the min path cost with infinity
2: u ← nil                    ▷ Init the new parent with nil
3: for all (v_i, (v_i, v) ∈ E) do          ▷ For all neighbors of v
4:    if (ISCLOSED(v_i)) then
5:       if (g(v_i) + |v_i − v| < minPathCost) then
6:          minPathCost ← g(v_i) + |v − v_i|     ▷ Update min cost
7:          u ← v_i                              ▷ Remember v_i as new parent
8:       end if
9:    end if
10: end for
11: if (u != nil) then
12:    SETPARENT(u, v)                          ▷ Set u as new parent of v
13:    g(v) ← g(u) + |v − u|                    ▷ Update path cost g
14:    f(v) ← g(v) + h(v)                       ▷ Update priority f
15:    PUSH(v, q)                               ▷ Push v into the priority queue
16: end if
17: return
```

Algorithm 3 Connect Obstacle

Input: Polygon $\boldsymbol{p}$

```
1: for all (v_i in VERTICES(p)) do          ▷ For all corners v_i of p
2:    if (isConvex(v_i)) then               ▷ If the corner v_i is convex
3:       V ← V ∪ v_i                        ▷ Add the corner v_i to the graph
4:       for all (v_j ∈ V, j ≠ i) do        ▷ For all known vertices v_j
5:          if (isTangential(v_i, v_j)) then
6:             E ← E ∪ (v_i, v_j)          ▷ Make v_i a neighbor of v_j
7:          end if
8:       end for
9:       FINDPARENT(v_i)
10:   end if
11: end for
12: return
```

C. Time Complexity

The time complexity of the Minimal Construct algorithm can be analyzed as follows. A* is known to have an asymptotic complexity of $O(n^2 \log n)$ operations if it is operating on a fully connected graph using a binary heap to implement the priority queue. Since in Algorithm 3 we connect (almost) every corner of a discovered obstacle with every known vertex in the graph, we do have to assume a fully connected graph. In addition to the A* search, our algorithm performs line intersection tests (Line 11 of Alg. 1), tangential tests (Line 5 of Alg. 3), parent finding (Line 32 of Alg. 1 and Line 9 of Alg. 3), and connecting obstacles (Line 34 of Alg. 1). We assume that deleting and creating parent and neighbor relationships can be done in constant time. This is for example possible with an adjacency matrix.

A* pops each vertex at most once from the queue, so we know that the while loop in line 8 of Algorithm 1 is executed at most $O(n)$ times. Consequently, a line intersection test is performed $O(n)$ times, each of which taking $O(n)$ operations, summing up to $O(n^2)$ in total.

The ConnectObstacle() routine is called $O(n)$ times in line 34 of Algorithm 1. Regardless, it loops only once over every vertex and connects each of them with $O(n)$ neighbors after performing a tangential test in lines 4, 5 and 6 of Algorithm 3. The tangential test and creating a neighbor relation are both in $O(1)$. Thus, the ConnectObstacle() function contributes $O(n^2)$ operations in total.

Finding a parent in Algorithm 2 performs constant time operations for $O(n)$ neighbors of a vertex v. The FindParent() routine is called at most once for every vertex in Line 32 of Algorithm 1 inside the while loop starting in line 8, and again at most once for every vertex inside the outer for loop starting in line 1 of Algorithm 3. These two occurrences amount to $O(n^2)$ operations spent on parenting in total.

In conclusion, all additional operations performed by Minimal Construct on top of the A* search are of amortized $O(n^2)$ complexity. This means that the A* search dominates the runtime of the algorithm with $O(n^2 \log n)$. The Overmars Welzl algorithm [4] has the same upper bound with $O(n^2)$ for the graph construction dominated by $O(n^2 \log n)$ for the search for the shortest path, but with a closer look we find that the construction of the entire Visibility Graph also has a lower bound of $\Omega(n^2)$. The lower bound of the Minimal Construct algorithm, however, is $\Omega(n)$. This is the

case when there is a line-of-sight between the start and the target and the line intersection test needs to be performed only for the single edge between s and t. Consequently, with both approaches having the same upper bound, but finding the shortest path only having a smaller lower bound than constructing the entire graph, the Minimal Construct approach must statistically find the shortest path in less time. This observation is supported by the systematic experiments presented in the next section.

IV. Experimental Results

We evaluated the performance of the Minimal Construct algorithm with respect to the Overmars Welzl algorithm [4] and also with respect to the grid-based A* and Lazy Theta* algorithms. We made effort to implement each of these algorithms as efficiently as possible by using code profiling techniques to improve computation time and reusing allocated memory for subsequent runs. Most notably, since the Overmars Welzl algorithm does not actually need to compute line intersections, it only needs to decide whether a point lies in front of or behind the nearest visible edge, we allowed it to exploit this advantage by using a fast scalar product-based test instead of computing actual line intersections or using a square root to determine a distance. A* and Lazy Theta* have been implemented using integer arithmetic for state transitions. The line-of-sight tests that Lazy Theta* needs are computed with the Bresenham algorithm.

The experiments were performed in two artificially generated maps of different nature. The maps are shown in Figure 3. The first type of map has a fixed size of 100x100 meters and includes a parameterized number of non-convex obstacles. By systematically increasing this parameter and adding more and more obstacles to the map, we were able to explore the runtimes of the aforementioned algorithms with respect to the number of polygon edges present in the environment. The maps shown in Figure 3 on the left have a medium degree of clutter with approximately 1500 edges. As can be seen in the plot on the left of Figure 4, Minimal Construct outperforms the other algorithms by a large margin. Up to approximately 2000 polygon edges in the map, Minimal Construct finishes in under a millisecond (runtimes are measured by averaging 1000 randomly chosen start and target combinations in the map). In this environment, Minimal Construct is able to exploit its full potential by adding only a relatively low number of obstacles into the graph and thus avoiding the vast majority of the line intersection tests that the Overmars Welzl algorithm has to perform. The grid-based algorithms are not aware of the number of obstacles in the scene. They search for a path in an occupancy grid that represents the obstacles as closely as possible. We used a cell size of 10 cm for the grid. The runtimes of the grid-based searches stay more or less constant, no matter how many obstacles are present.

The second map was designed to predict how well these algorithms would perform in an indoor environment. Mimicking an office building, we generated a map with square shaped rooms that are laid out on both sides along a corridor. The corridor runs in a loop around a square shaped building. The rooms have a wall length of 3 meters. This time, the varied parameter is the number of rooms along one side of the building. The smallest building has a size of 21x21 meters with seven rooms along one side of the building. The map shown in Figure 3 on the right has 12 rooms along one side. We evaluated maps with up to 50 rooms and a 150 square meters large footprint, even though the actual space inside the building is much smaller since the inner yard of the building is not used. This environment represents a hard case for the Minimal Construct algorithm, because the whole interior ring of rooms is one large polygon that gets in the way of nearly every path while contributing a large number of edges to iterate over when computing a line intersection test. The bounding box acceleration of the line intersection checks is not effective for such large obstacles that intersect with almost every line. Even so, by delaying the line intersection tests, Minimal Construct is still able to reliably outperform the entire construction of the Visibility Graph. The grid-based searches A* and Lazy Theta* perform worse for smaller versions of the Office map, but their runtime is less sensitive to the size. Starting from buildings with a size of 30 rooms per side, a standard A* search begins to find paths in less time than Minimal Construct. Real world maps can perhaps be broken up into smaller polygons and topological mapping [19], [20] could be used in order to reduce the size of the map that needs to be searched for a shortest path in order to maintain the upper hand of a polygonal map representation for large environments.

V. Conclusions

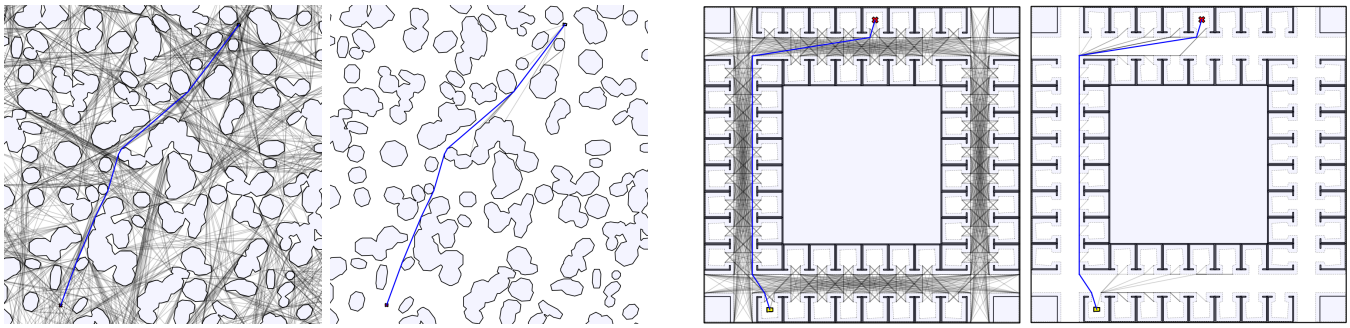
We presented a novel, Visibility Graph-oriented path search algorithm that significantly reduces the computation time of finding the shortest path in a polygonal map. With the philosophy of not constructing the entire Visibility Graph, but only a necessary portion, the Minimal Construct algorithm is consistently able to compute shortest paths in cluttered and indoor-like environments faster than one of the best known $O(n^2)$ Visibility Graph construction algorithms combined with an A* search. Our algorithm performed well even in comparison to standard, eight-connected A* grid searches.

Polygonal planning and modeling comes with the added benefit of a non-discrete action set and continuous world representation that yields smooth and optimal shortest paths with a minimum number of path segments. We envision to gain further benefits of a polygonal representation in future work in the context of dynamic motion planning by leveraging mathematical expressions to predict collision points by intersecting motion trajectories with edges of possibly moving polygons.

References

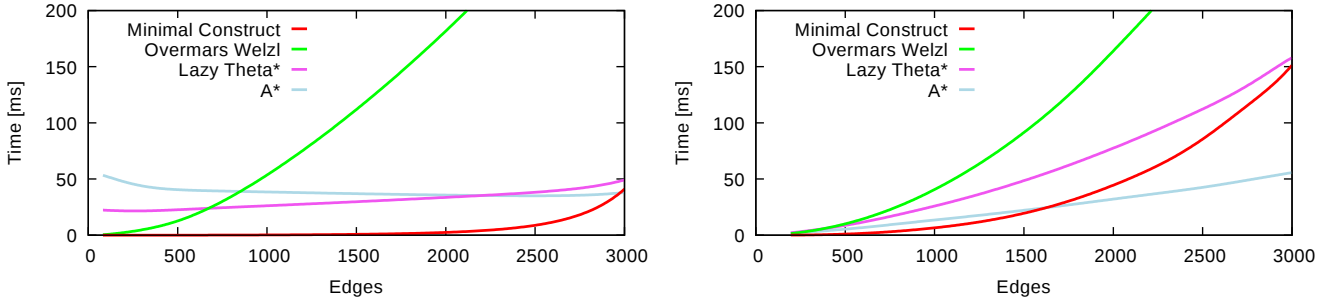
[1] E. Marder-Eppstein, E. Berger, T. Foote, B.P. Gerkey, and K. Konolige. The office marathon: Robust navigation in an indoor office environment. In *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, 2010.

[2] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Robotics*, 2009.



(a) Overmars Welzl (b) Minimal Construct (c) Overmars Welzl (d) Minimal Construct

Fig. 3: The maps that have been used to evaluate our shortest path finding algorithms in comparison to a full construction of the Visibility Graph. The two images on the left show a cluttered type of map where obstacles are randomly scattered in a 100x100 meters large square. The two images on the right show an office building that we generated. a) An example path found in the fully constructed Visibility Graph. The collision-free edges of the Visibility Graph are shown with grey lines. b) The identical path found by the Minimal Construct algorithm. Collision-free graph edges that were tested for line intersection are shown with grey lines. c) The full Visibility Graph of the Office map and the shortest path found within that graph. d) The same path found by our Minimal Construct algorithm and the collision-free edges that were tested for line intersection.



(a) Clutter (b) Office

Fig. 4: Average runtimes in a) the Clutter map and b) the Office map with increasing map complexity in terms of polygon edges. In the Clutter map, the Minimal Construct algorithm is able to exploit its full potential and computes shortest paths much faster than the other algorithms. The Office map is a hard case for our algorithm and it performs worse, even though it still finds shortest paths significantly faster than the full construction of the Visibility Graph and faster than the grid based searches up to a building size of approximately 30 rooms per line or 150 m^2.

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

[4] M. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proc. of the Symposium on Computational Geometry (SCG)*, pages 164–171, 1988.

[5] N.J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In *Proc. of the Int. Conf. on Artificial Intelligence (IJCAI)*, pages 509–520, 1969.

[6] D.T. Lee. *Proximity and Reachability in the Plane*. Dissertation, University of Illinois at Urbana-Champaign, 1978.

[7] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility of disjoint polygons. *Algorithmica*, 1(1):49–63, 1986.

[8] E. Welzl. Constructing the visibility graph for n-line segments in o(n^2) time. *Information Processing Letters*, 20(4):167 – 171, 1985.

[9] K.G. Subir and M.M. David. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing (SICOMP)*, 20(5):888–910, 1991.

[10] J. Kitzinger. *The Visibility Graph Among Polygonal Obstacles: A Comparison of Algorithms*. University of New Mexico, 2003.

[11] H. Rohnert. Shortest paths in the plane with convex polygonal obstacles. *Information Processing Letters*, 23(2):71–76, 1986.

[12] H.-P. Huang and S.-Y. Chung. Dynamic visibility graph for path planning. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (IROS)*, 2004.

[13] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[14] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime search in dynamic graphs. *Artificial Intelligence*, 172(14):1613 – 1643, 2008.

[15] S. Koenig and M. Likhachev. D^* Lite. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2002.

[16] D. Ferguson and A. Stentz. *Field D^*: An Interpolation-Based Path Planner and Replanner*, pages 239–253. Springer Verlag, 2007.

[17] A. Nash, S. Koenig, and M. Likhachev. Incremental Phi*: Incremental any-angle path planning on grids. In *Proc. of the Int. Conf. on Artificial Intelligence (IJCAI)*, 2009.

[18] A. Nash, S. Koenig, and Craig A. Tovey. Lazy Theta*: Any-angle path planning and path length analysis in 3D. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2010.

[19] F. Blöchliger, M. Fehr, M. Dymczyk, T. Schneider, and R. Siegwart. Topomap: Topological mapping and navigation based on visual SLAM maps. *CoRR*, abs/1709.05533, 2017.

[20] Sebastian Thrun. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21 – 71, 1998.