ACCELERATING PATH PLANNING WITH VECTORIZATION OF INTERSECTION OPERATIONS

Kasmynin Kirill, Mironov Konstantin
Moscow Institute of Physics and Technology
Moscow, Russia
kirill.kas@mail.ru

ABSTRACT

Path planning in environments with obstacles is addressed through the construction of visibility graphs, known for high accuracy but computationally expensive edge intersection checks. A fully vectorized approach is introduced to optimize intersection-checking operations, significantly reducing computation time by processing multiple edges simultaneously. Performance improvements are demonstrated through experiments, showing a substantial reduction in graph construction time compared to traditional and partially vectorized methods. Integration with various polygon extraction techniques is explored, and comparisons are made with another pathfinding algorithms. Validation on large, complex environments highlights the scalability and efficiency of the proposed approach for navigation tasks in mobile robotics.

***K*eywords** Path planning, visibility graph, vectorization, mobile robots, intersection checks, polygon extraction, computational efficiency, obstacle detection

1 Introduction

The problem of planning an optimal route in environments with obstacles is a relevant and challenging research direction [1].

One of the key tasks in autonomous robotics is path planning for mobile robots in environments with obstacles. Mobile robots are used for a variety of applications, including delivery services, performing household tasks, and exploring hard-to-reach locations.

Path planning is one of the most critical aspects of navigation processes. It involves determining a collision-free path in a given environment. Path planning systems must construct routes from the start point to the end point safely and efficiently, considering factors such as time, distance, energy, and other constraints [2].

Path planning algorithms aim to determine the most efficient path or a reasonable approximation, optimizing one or more objective functions. For example the path could minimize travel time, which is crucial for tasks like search and rescue operations, or reduce energy consumption, ensuring the robot's resources are used efficiently. Alongside finding an effective path, the ability to plan it quickly is equally critical [3].

The concept of visibility graphs, which has been around for quite some time, provides a way to solve this problem. Visibility graphs [4] offer a pathfinding solution, but their construction and the extraction of obstacle polygons from maps pose significant computational challenges.

Visibility graphs [4] rely on constructing a graph where nodes represent obstacle's vertices and edges connect visible vertices, ensuring collision-free paths.

One of the most computationally difficult stages in visibility graph construction is the intersection-checking process. This involves determining whether edges connecting nodes intersect with obstacle boundaries. The main problem is to avoid pairwise comparison of a large number of possible visibility edges and obstacle edges, which leads to nested

loops and significant performance overhead. To solve this problem the proposed approach uses vectorization techniques to perform intersection checks efficiently, reducing dependence on iterative cycles and minimizing computational costs.

It is necessary to ensure accurate detection and appropriate approximation of polygons extracted from the occupancy grid or image, whether using image processing techniques or neural networks, like PolyWorld [5] or by first segmenting obstacles with, e.g., U-Net to obtain a binary mask and then extracting polygonal contours. The correct representation of polygons is crucial for the efficient and accurate construction of a visibility graph, which plays a fundamental role in path planning.

Using polygon approximation techniques and vectorizing intersection-checking operations, the process of visibility graph construction can be significantly accelerated. These optimizations reduce computational overhead and enable the rapid generation of collision-free paths.

In previous research [6], polygon extraction based on neural network methods combined with partial vectorization to enhance performance. Although these approaches accelerated the construction of visibility graphs, there were still opportunities for further acceleration. This study introduces a shift towards full vectorization, enabling significantly faster and more efficient intersection checks. Additionally, we focus on classical methods for polygon approximation and extraction, which reduce computation time while maintaining the precision required for path planning. The work also expands beyond image-based methods by directly utilizing occupancy grids, ensuring better compatibility with robotic navigation systems.

Path planning in environments with obstacles relies on various algorithms to determine efficient routes. This research focuses on visibility graphs combined with accurately defined and effectively approximated polygons. Accurate polygon construction important for efficient use of visibility graphs. The integration of these graphs with vectorized intersection operations addresses challenges related to computational efficiency and time consumption path planning. Additionally, widely used algorithms such as A* [7] and Theta* [8] are included in the comparative analysis of different approaches.

2 Formal Problem Statement

The problem considered in this research involves constructing a visibility graph for path planning in environments with obstacles. The primary goal is to identify obstacles and represent them as polygons to facilitate efficient and accurate graph construction.

Given a map I, the objective is to determine the set of obstacles $O = \{O_1, O_2, ..., O_n\}$ on that map.

The output of the algorithm is the set of polygons $P = \{P_1, P_2, ..., P_m\}$ and a visibility graph $G(V, E)$.

The problem considered in this research involves finding the shortest path on visibility graph.

Let $\mathcal{E}$ represent the environment, where:

- $\mathcal{E}_{\text{free}} \subset \mathcal{E}$ is the navigable free space.
- $\mathcal{E}_{\text{obs}} \subset \mathcal{E}$ represents the obstacle space.

These subsets satisfy $\mathcal{E} = \mathcal{E}_{\text{free}} \cup \mathcal{E}_{\text{obs}}$ and $\mathcal{E}_{\text{free}} \cap \mathcal{E}_{\text{obs}} = \emptyset$.

The task is to construct a visibility graph, $G = (V, E)$, where V represents the set of vertices. This set includes the start position s, the goal position g, and the vertices of obstacle polygons V_{poly}, such that $V = \{s, g\} \cup V_{\text{poly}}$. The set E consists of edges that connect vertices without intersecting any obstacles in $\mathcal{E}_{\text{obs}}$.

The primary computational challenge in constructing G lies in intersection checking, which involves determining whether a candidate edge $e = (v_i, v_j)$ intersects with any obstacle polygon.

$$\text{Intersect}(e, \mathcal{E}_{\text{obs}}) = \begin{cases} \text{True}, & \text{if } e \cap \mathcal{E}_{\text{obs}} \neq \emptyset, \\ \text{False}, & \text{otherwise.} \end{cases}$$

To address this challenge, the proposed approach includes the following:

- Extracting obstacle polygons $P = \{P_1, P_2, \ldots, P_n\}$ from $\mathcal{E}_{\text{obs}}$, where each P_i is a polygon defined by its vertices $\{p_1, p_2, \ldots, p_k\}$.
- Approximating the polygons to simplify computation while ensuring accuracy for visibility graph construction.
- Fully vectorizing intersection-checking operations to eliminate iterative pairwise comparisons and reduce computational cost.

The visibility graph is defined as:

$$G = (V, E), \quad \text{where } E = \{(v_i, v_j) \mid v_i, v_j \in V, \text{Intersect}((v_i, v_j), P_k) = \text{False } \forall k\}.$$

2.1 Performance Metrics

To evaluate the proposed algorithm, several metrics are considered. The first metric is computational complexity, which measures the time required to find a path. The second metric is path quality, which evaluate the length of the generated path. Lastly, scalability is evaluated, reflecting the algorithm's ability to work with larger maps and environments.

This work extends previous research by incorporating full vectorization of intersection-checking operations and different methods for polygon extraction and approximation. These advancements aim to improve the efficiency of visibility graph construction.

3 Related works

3.1 Path planning methods

Path planning is a critical component in the motion control of mobile robots, enabling navigation from an initial position to a goal while considering for obstacles [9]. The process involves defining a sequence of states that guide the robot from its current position to the goal [10]. Achieving the goal state requires compliance with physical constraints such as time and collision avoidance. Also path evaluation often involves optimizing criteria such as minimizing travel time or path length [10].

Path planning methods can be broadly categorized based on the environment. In static environments, the configuration remains unchanged during navigation, while in dynamic environments, the robot must adapt to changing obstacles [11]. Additionally, these methods can be classified based on the availability of environmental information: global path planning assumes complete knowledge of the environment, whereas local path planning operates under partial or incomplete information, requiring real-time updates [9].

Various approaches are employed in path planning, including graph-based methods, cellular decomposition methods, potential field methods and techniques based on artificial intelligence technologies. Many of these methods yield a chain of waypoints connecting the start and end points of the path [9].

In path planning solutions that employ methods of grid-based decomposition or graphs, an stage of pathfinding on the graph is often required [9].

Classic search strategies include methods such as breadth-first search, cost-based search, depth-first search, limited depth search, depth-first search with iterative deepening, and bidirectional search. These strategies are described in the current work [12]. They are based on heuristic rules specific to the given task, enhancing the algorithms efficiency compared to classical strategies.

An example of a heuristic strategy is the A* algorithm [7], which is an extension of Dijkstra's algorithm [13].

Algorithms based on cell decomposition face limitations, particularly the restricted directional movement imposed by the grid structure. To address these issues, methods such as the D* algorithm [14] and Theta* algorithm [8] have been proposed.

Probabilistic roadmap (PRM) methods offer an alternative graph-based approach in which a sparse set of collision-free configurations is first sampled in the free space, then connected by local planners [15].

Were reviewed numerous methods applicable for solving tasks in path planning.In summary, path planning methods play a crucial role in mobile robot motion control, facilitating navigation through obstacles from an initial to a goal point.

3.2 Image Processing Techniques and Polygon Approximation

Image processing techniques can extract obstacle polygons from binary images by analyzing edge points and their topological distribution to form enclosed polygons [16]. Such binary images can be generated from occupancy grid maps, representing obstacles and free space in a structured format.

Polygon approximation reduces the number of vertices in extracted polygons while maintaining their shape by analyzing local curvature and inner angles between edges [17].

In addition to processing binary images, the initial step often involves obtaining these images through segmentation models or other preprocessing techniques to identify obstacles accurately.

An effective neural network method for extracting polygons from images is described in [5]. PolyWorld is a neural network designed to directly extract building vertices from images and connect them into polygons.

These methods aim to accurately represent obstacles in environments for efficient path planning algorithms.

4 Methodology

This section outlines the methodology for constructing visibility graphs efficiently. The approach focuses on full vectorization of intersection-checking operations, leveraging both high-performance computing techniques and effective graph construction strategies.

4.1 Vectorization of Intersection Operations

Constructing a visibility graph involves checking intersections between obstacle edges and potential graph edges. This process can become computationally expensive due to the large number of pairwise checks required. To address this, we implemented full vectorization for intersection checking.

Held's method [18] is employed to decide intersection of two line segments AB and CD in a collision-free manner. For any three points $a = (a_x, a_y)$, $b = (b_x, b_y)$ and $p = (p_x, p_y)$, the determinant is defined as:

$$\det(a, b, p) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ p_x & p_y & 1 \end{vmatrix}$$

A point p lies strictly to the left of the directed line from a to b if $\det(a, b, p) > 0$. Two segments AB and CD intersect precisely when

$$\det(A, B, C)\ \det(A, B, D) < 0 \quad \text{and} \quad \det(C, D, A)\ \det(C, D, B) < 0.$$

In the fully vectorized implementation, the four arrays of determinants are computed in parallel for all candidate edge pairs, producing a Boolean mask that identifies non-intersecting edges.

In the vectorized approach, all obstacle edges and potential graph edges are represented as NumPy arrays, allowing for efficient processing. Instead of performing sequential checks for every pair of edges, the method uses advanced array operations to compute intersections simultaneously for all edges.

1. The coordinates of obstacle edges are repeated for every graph edge using `np.tile`, which duplicates the obstacle edge data N times, where N is the number of potential graph edges. This ensures that every graph edge can be paired with all obstacle edges.

2. Similarly, the coordinates of potential graph edges are repeated for every obstacle edge using `np.repeat`, which duplicates the graph edge data M times, where M is the number of obstacle edges. This enables the creation of all possible edge combinations.

3. Determinants are computed for these pairs of edges by processing them as single vectors. These determinants evaluate the relative position of endpoints to determine if the edges intersect. All computations are performed directly on the vectors, avoiding the need for explicit loops. This approach ensures efficient processing by leveraging the inherent advantages of vectorized operations in NumPy, effectively avoiding iterative checks.

4. The intersection results are then reshaped into a two-dimensional array, where each entry corresponds to the intersection status of a specific edge pair. This array is collapsed into a boolean mask identifying visible edges—those that do not intersect with any obstacle.

The vectorized approach removes the need for nested loops, significantly accelerating the intersection-checking process. The output is a set of edges that are free from intersections with obstacles. The Python implementation leverages NumPy for array manipulations and determinant calculations, ensuring high performance.

Below is the pseudocode summarizing the vectorized process:

procedure ISEDGEINTERSECTING($edges$)
    $N \leftarrow$ number of $edges$
    $M \leftarrow$ number of $obstacle_edges$
    $obstacle_edges \leftarrow$ array of obstacle edges
    $all_edges \leftarrow$ array of edges

$x1 \leftarrow \text{Tile}(obstacle_edges[:, 0], N)$
$y1 \leftarrow \text{Tile}(obstacle_edges[:, 1], N)$
$x2 \leftarrow \text{Tile}(obstacle_edges[:, 2], N)$
$y2 \leftarrow \text{Tile}(obstacle_edges[:, 3], N)$
$x3 \leftarrow \text{Repeat}(all_edges[:, 0], M)$
$y3 \leftarrow \text{Repeat}(all_edges[:, 1], M)$
$x4 \leftarrow \text{Repeat}(all_edges[:, 2], M)$
$y4 \leftarrow \text{Repeat}(all_edges[:, 3], M)$
$D_1 \leftarrow (x1 \cdot y2 + x2 \cdot y3 + x3 \cdot y1) - (y1 \cdot x2 + y2 \cdot x3 + y3 \cdot x1)$
$D_2 \leftarrow (x1 \cdot y2 + x2 \cdot y4 + x4 \cdot y1) - (y1 \cdot x2 + y2 \cdot x4 + y4 \cdot x1)$
$D_3 \leftarrow (x3 \cdot y4 + x4 \cdot y1 + x1 \cdot y3) - (y3 \cdot x4 + y4 \cdot x1 + y1 \cdot x3)$
$D_4 \leftarrow (x3 \cdot y4 + x4 \cdot y2 + x2 \cdot y3) - (y3 \cdot x4 + y4 \cdot x2 + y2 \cdot x3)$
$intersects \leftarrow (sign(D_1) \cdot sign(D_2) < 0) \wedge (sign(D_3) \cdot sign(D_4) < 0)$
$intersects \leftarrow \text{Reshape}(intersects, (N, M))$
$visible_edges_mask \leftarrow \neg\text{any}(intersects, \text{axis} = 1)$
$visible_edges \leftarrow$ Filter $edges$ using $visible_edges_mask$
return $visible_edges$
end procedure

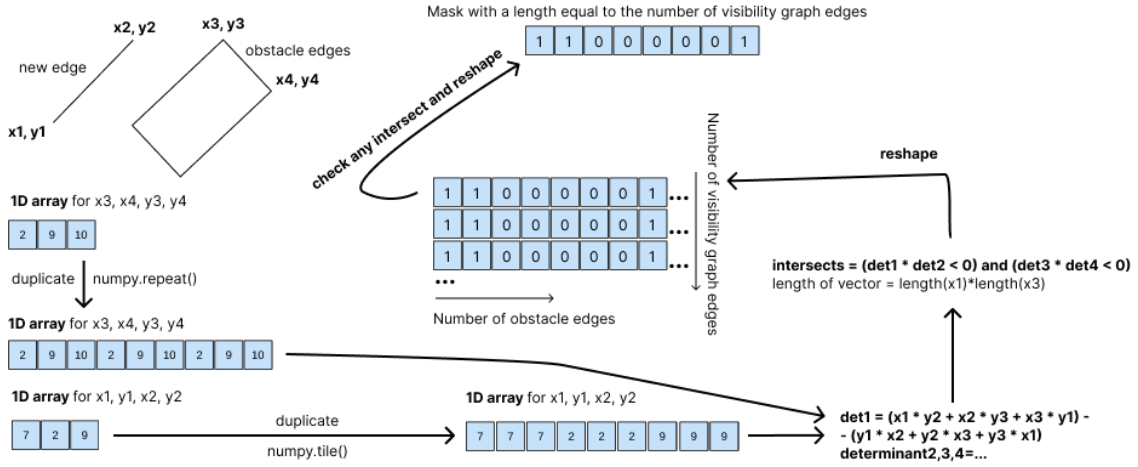

Figure 1: Representation of vectorized intersection check operations

4.2 Implementation Approaches: AVX and NumPy

Two main implementation strategies can be utilized for optimizing intersection operations:

NumPy-based implementation: Leveraging NumPy arrays, this approach is platform-independent and efficient for environments where AVX instructions are unavailable. NumPy handles array operations and determinant calculations, enabling quick and scalable vectorized computations.

AVX-based implementation: Advanced Vector Extensions (AVX) enable significant acceleration by processing multiple floating-point operations simultaneously using SIMD (Single Instruction, Multiple Data) architecture. This method is particularly useful for large-scale intersection checks on compatible CPUs.

Both strategies are viable for visibility graph construction, with NumPy offering simplicity and flexibility, while AVX can provide additional speedups in hardware-supported environments. Experiments with AVX implementations can further evaluate its potential benefits in reducing computation time.

Modern CPUs support SIMD (Single Instruction, Multiple Data) instructions, which allow a single operation to be applied to multiple data elements in parallel. SIMD using instruction sets such as AVX, which allow operations to be performed simultaneously on vectorized data. Significant performance improvements can be achieved compared to standard implementations [19]. In the context of intersection operations, SIMD enables checking multiple edge pairs simultaneously rather than sequentially. This is useful for constructing visibility graphs, where the number of potential intersections increases quadratically with increasing number of vertices.

However, in order to use SIMD effectively, calculations must be explicitly reformulated to work in parallel with graph edges. This is made possible by converting the coordinates of all edges into vector representations. Instead of calculating pairwise intersections in nested loops, arrays of edge endpoints are created and stored as NumPy arrays or AVX-aligned arrays.

This vectorized formulation allows for two implementation strategies. The first is a NumPy-based implementation in which intersection calculations are performed using NumPy functions. NumPy is built on optimized C interfaces, and many operations are performed using SIMD instructions. Standard arithmetic operations are already supported by the built-in AVX functions [20] . Thus, vectorized NumPy code can provide execution speeds comparable to manually written AVX code.

The second strategy uses manual AVX vectorization, in which segment data is explicitly loaded into AVX registers (for example, types m256), and all operations are implemented using built-in AVX functions. This approach provides more precise control over data alignment and performance at the register level.

Advanced Vector Extensions (AVX) represent a set of SIMD (Single Instruction, Multiple Data) instruction set extensions introduced by Intel to enhance the performance of compute-intensive applications. AVX increases the width of the vector registers (up to 256 bits in AVX and even wider in AVX-512), allowing parallel processing of multiple data elements per instruction. This capability enables speedups in arithmetic operations. AVX reduces loop overhead and maximizes throughput, which makes it a useful technology for solving high-performance computing tasks [21].

Both the NumPy-based implementation and the AVX-based implementation provide efficient vectorization of intersection operations. This significantly speeds up the construction of the visibility graph. The proposed approach converts the coordinates of the boundaries into vector representations, allowing efficient use by the implementation using NumPy and AVX.

The planned experiments will allow us to compare these approaches to implementation in different conditions in order to assess their efficiency.

4.3 Visibility Graph Construction

Graph-based algorithms are extensively applied in path planning tasks [22]. Notable examples include the visibility graph algorithms [23], Voronoi diagrams [24], rapidly exploring random trees (RRT) [25], and probabilistic roadmaps [15].

The visibility graph is constructed by first extracting obstacle polygons and their vertices. Using the fully vectorized intersection-checking method described above, edges connecting all vertices, including start and goal points, are validated. Edges that do not intersect obstacles are added to the graph.

The visibility graph is constructed by first creating a complete graph where visibility edges connect all obstacle vertices. Edges are filtered to include only those that do not intersect with obstacles. Additional edges are added between the start and goal points to that graph. This approach enables the reuse of the constructed visibility graph for different goal points, reducing computation time for subsequent path planning tasks.

The methodology involves several key modules: obstacle extraction, visibility graph construction, shortest path computation, and visualization. Obstacle extraction prepares the input data, while the visibility graph construction handles the creation and filtering of edges. The Dijkstra's algorithm module identifies the shortest path, and the visualization module provides an analysis of the graph and the path.

A pseudocode representation and a detailed explanation of the visibility graph construction algorithm are provided in a previous work [6].

4.4 Advantages of Visibility Graphs for Path Planning

The visibility graph provides an approach to path planning by explicitly modeling the free space for navigation by connecting the vertices of obstacles. Unlike grid-based methods that divide the environment into cells, visibility graphs connect all mutually visible vertices of polygonal obstacles, as well as the start and end positions. The result is a graph structure in which each edge is a straight line without collisions.

One of the main advantages of this approach is the sparsity of the resulting graph. Since visibility graphs consist only of obstacle vertices and individual visibility edges, the number of nodes is significantly less than in grid-based representations, which can contain thousands of individual cells. This leads to a more efficient graph traversal at the pathfinding stage, for example, using Dijkstra.

In addition, the visibility graph provides the construction of shortest paths. Unlike methods based on a discrete grid, such as A*, which are limited by the direction of movement along the grid cells and often lead to inefficient trajectories, the visibility graph allows you to find optimal paths accurately avoiding obstacles.

Another important advantage is the separation of the graph construction stage and the pathfinding stage. Once the visibility graph is built, it can be reused for multiple queries. This makes the approach particularly effective in replanning scenarios where only the start or end points are changed. In such cases, the graph structure remains relevant, and it is only necessary to recalculate the shortest path, which is performed in milliseconds. In contrast, other methods are forced to recalculate significant parts of their internal structure with each new request.

Visibility graphs, with their sparse structure, high accuracy, and the ability to quickly reschedule, provide a high-performance and scalable solution to the problem of finding a path in environments with obstacles. The following section provides a description of the implemented replanning mechanism.

4.5 Replanning Mechanism

In many robotics applications, the start or target may change during execution, which requires a quick recalculation of the trajectory. To solve this problem, a replanning strategy has been developed that significantly reduces the replanning time in the same environment, using an already created visibility graph and supplementing it with only new start and end vertices and their edges.

Replanning capability is only achievable in graph-based planners—such as the visibility graph and PRM—where the prebuilt roadmap can be incrementally augmented, whereas grid-based algorithms (e.g. A*, Theta*) must reprocess the entire grid for each new query.

To implement replanning, it is necessary to first build a single basic visibility graph based on obstacle polygons, which is saved for later use. This initial graph, defined by a set of vertices V_0 (all obstacle angles) and a set of edges E_0 (all pairwise visibility edges), reflects the complete topology of the free space of the environment. Once this basic graph is constructed and stored in memory, it will not need to be reconstructed unless the geometry of the obstacle change.

When a specific start–goal query (s, g) is presented, the saved graph is augmented by inserting the two new vertices s and g and then generating only those edges incident to them which do not have collisions with obstacles. Formally:

$$V = V_0 \cup \{s, g\}, \qquad E = E_0 \ \cup \ \{(s, v) \mid v \in V_0, \ \text{visible}(s, v)\} \ \cup \ \{(g, v) \mid v \in V_0, \ \text{visible}(g, v)\}.$$

A single execution of Dijkstra's algorithm on (V, E) then produces the updated shortest path in milliseconds. If subsequently only the goal or start changes, the same procedure is repeated against the unchanged base graph, ensuring that each replanning request incurs minimal overhead.

4.6 ROS Node Implementation

The visibility-graph planner has been implemented as a ROS node. The node subscribes to the occupancy grid (e.g. `/occupancy_grid_map/grid_map`), converts it into a binary image, and filters out small noisy regions below a configurable area threshold. Contours are then extracted via OpenCV's `findContours` and simplified using the Douglas–Peucker algorithm [17] to create polygonal obstacle contours. An additional filter at the vertex level removes points whose pairwise distance is less than the minimum, providing a sparse graph without loss of accuracy.

Polygon vertices are transformed from pixel indices into metric coordinates using the grid's resolution and origin. From these vertices, the base visibility graph is built once per map with our vectorized intersection operations. At each replanning request the graph is extended by inserting the current start and goal points. ROS node receive the current robot pose and a user-specified goal from RViz or scripts are added as new vertices, and their visibility edges against the stored vertex set are computed. Finally, Dijkstra's algorithm is executed on this augmented graph to find path.

The resulting waypoints are published as a `nav_msgs/Path` for controllers and displayed in RViz.

5 Experiments and Results

Experiments were conducted to compare various path-planning algorithms, including the Visibility Graph, Theta*, and A*, under identical conditions. The environment was represented as a list of obstacle vertices extracted from a 300x300 image. The results include the execution time for environment setup (including polygon extraction) and pathfinding. The experimental results are shown in Table 1.

Experiment	Method	Execution Time
1	Polygons without approximation + Visibility Graph + Dijkstra	170-200 seconds
2	Polygons with approximation + Visibility Graph + Dijkstra	6.4 seconds
3	Partial Vectorization + Polygons with approximation + Visibility Graph + Dijkstra	1.53 seconds
4	Partial Vectorization + Polygons without approximation + Visibility Graph + Dijkstra	36.05 seconds
New Full Vectorization		
5	Full Vectorization + Polygons with approximation + Visibility Graph + Dijkstra	**0.05** seconds
6	Full Vectorization + Polygons without approximation + Visibility Graph + Dijkstra	2 seconds
7	Theta*	6.286 seconds
8	A*	50.1 seconds

Table 1: Environment Construction and Pathfinding Time

Full vectorization significantly reduces execution time compared to previous approaches.

Polygon simplification reduces visibility-graph construction from roughly 200 s to 6.4 s (Experiment 2). Introducing partial vectorization of the edge-intersection tests cuts that further to 1.53 s when used together with simplification (Experiment 3). Most importantly, our new full vectorization of those same intersection checks drives build time down to 0.05 s on approximated obstacles (Experiment 5). These results show that first reducing polygon complexity and then fully vectorizing the most expensive operations is an effective strategy for visibility graph path planning.

Another experiment on a larger map. To evaluate the scalability and efficiency of the visibility graph approach compared to Theta*, an additional experiment was conducted on a larger, obstacle-dense map. The results demonstrated the significant advantages of the visibility graph method in handling complex environments. The Theta* algorithm required approximately 180 seconds to construct the path. The visibility graph completed in 4.9 seconds. Also once the visibility graph was constructed, the actual pathfinding process took only 1 millisecond.

5.1 Comparison with Probabilistic Roadmaps (PRM)

Probabilistic Roadmaps (PRM) construct a connectivity graph by randomly sampling the free space. In contrast, the visibility graph is constructed using the vertices of the obstacle polygons, which guarantees the shortest path. To achieve similar accuracy using PRM, it is necessary to increase the number of samples, which leads to a rapid increase in the number of edges and an increase in computational costs.

In experiments compare such indicators as graph construction time and path quality. Figure plots the build time against path-length error for both methods on small and large maps. The visibility graph always finds a 100%–optimal route: on the small map it builds in just 50 ms, and on the large map in 4 091 ms. By contrast, PRM's build time and error vary with sample count and k:

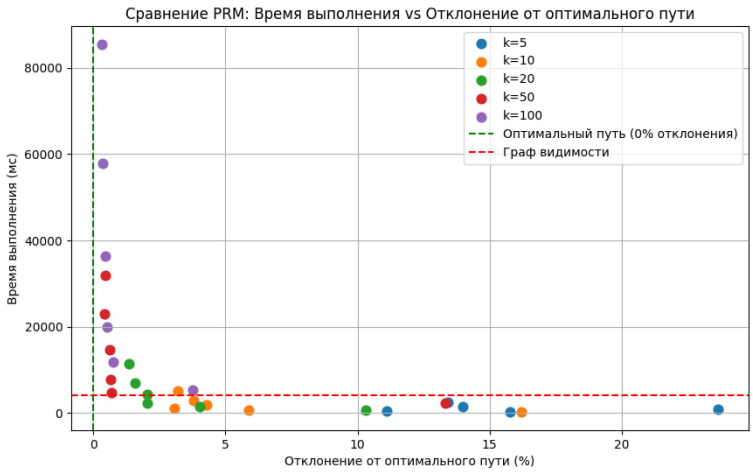

Figure 2: Build time and path-length deviation

Table 2: PRM build times and path-length deviations compared to the visibility graph

Method	Build Time (ms)	Path Length	k
Visibility Graph (small map)	50	**100%**	—
PRM, 50 samples	80	117%	10
PRM, 100 samples	112	107%	10
PRM, 65 samples	223	110%	20
Visibility Graph (large map)	4091	**100%**	—
PRM, 500 samples	359	104–111%	10
PRM, 1 000 samples	1289	102–107%	15
PRM, 5 000 samples	3128	104–106%	10
PRM, 5 000 samples	4574	102–103%	15

These results confirm that, for paths requiring true optimality, the visibility graph outperforms PRM in both build time (on the small map) and path quality (on both maps). PRM can approach optimal lengths only with very large sample sets at the cost of much longer build times.

5.2 NumPy and AVX Performance

The fully vectorized intersection checking operations was implemented both in Python using NumPy and in C++ with AVX. On a small 300×300 map, the Python/NumPy implementation completed graph construction in 52 ms where the AVX-based C++ version required 104 ms. On a large, obstacle-dense map, the two implementations showed virtually identical runtimes: 4 091 ms for Python/NumPy and 4 119 ms for C++/AVX. These results indicate that, for angle based visibility graph construction, NumPy approach can match or even outperform a hand-optimized AVX implementation.

5.3 Replanning Performance

Replanning on the prebuilt visibility graph proved extremely fast. On a large, obstacle-dense map the initial visibility-graph construction required approximately 3.9 s, where subsequent replanning queries completed in roughly 34-37 ms. By comparison, a PRM roadmap replanning took 38–62 ms, but the paths were 1–3.4 % longer than optimal.

In a ROS–based test, visibility-graph construction took 2-3 s for large map and each replanning step averaged about 100 ms. These findings confirm that once the visibility graph constructed, replanning can be performed in a few milliseconds, supporting real-time route updates.

6 Conclusion

Experiments have shown that the algorithm successfully finds the shortest path on various sets of obstacles.

Experiments confirmed that simple polygon approximation further speeds up graph construction by reducing the number of vertices without sacrificing route optimality.

In this study, a fully vectorized approach for intersection operations in visibility graph construction was presented. The method significantly reduces computational complexity. Experiments demonstrated notable performance improvements. The results highlight the potential of vectorization techniques in accelerating path planning processes, particularly in large-scale and obstacle-dense environments.

References

[1] Fan Yang, Chao Cao, Hongbiao Zhu, Jean Oh, and Ji Zhang. FAR planner: Fast, attemptable route planner using dynamic visibility update. *CoRR*, abs/2110.09460, 2021.

[2] Karthik Karur, Nitin Sharma, Chinmay Dharmatti, and Joshua E Siegel. A survey of path planning algorithms for mobile robots. *Vehicles*, 3(3):448–468, 2021.

[3] Jose Ricardo Sanchez-Ibanez, Carlos J Pérez-del Pulgar, and Alfonso García-Cerezo. Path planning for autonomous mobile robots: A review. *Sensors*, 21(23):7898, 2021.

[4] 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.

[5] Stefano Zorzi, Shabab Bazrafkan, Stefan Habenschuss, and Friedrich Fraundorfer. Polyworld: Polygonal building extraction with graph neural networks in satellite images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1848–1857, 2022.

[6] Kirill Kasmynin and Konstantin Mironov. Vectorized visibility graph planning with neural polygon extraction. In *International Conference on Interactive Collaborative Robotics*, pages 265–280. Springer, 2024.

[7] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[8] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[9] V. Liu. Path planning methods in obstacle environment (review). *Mathematics and Mathematical Modeling*, (1):15–58, 2018.

[10] Vyacheslav Khasanovich Pshikhov, Mikhail Yuryevich Medvedev, Dmitry Olegovich Brosalin, Maria Aleksandrovna Vasilieva, Boris Viktorovich Gurenko, and Nizar Hamdan. Research on motion planning methods in two-dimensional mapped environments. *Bulletin of the Southern Federal University. Technical Sciences*, (3 (227)):170–192, 2022.

[11] Alexey Leonidovich Afanasov. Analysis of path planning methods for autonomous mobile devices. *Issues of Science and Education*, (17 (64)):9–17, 2019.

[12] A. Yahja et al. Framed-quadtree path planning for mobile robots operating in sparse environments. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, pages 650–655, 1998.

[13] EW DIJKSTRA. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] A. T. Sadiq, F. A. Raheem, and N. A. F. Abbas. Optimal trajectory planning of 2-dof robot arm using the integration of pso based on d* algorithm and cubic polynomial equation. In *First Int. Conf. Eng. Res.*, pages 458–467, 2017.

[15] L. E. Kavraki et al. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[16] Satoshi Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*, 30(1):32–46, 1985.

[17] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.

[18] Martin Held. Erit—a collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[19] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok-Ho Myung. Performance of sse and avx instruction sets. *arXiv preprint arXiv:1211.0820*, 2012.

[20] Oleksandr Pavlyk, Denis Nagorny, Andres Guzman-Ballen, Anton A Malakhov, Hai Liu, Ehsan Totoni, Todd A Anderson, and Sergey Maidanov. Accelerating scientific python with intel optimizations. In *SciPy*, pages 106–112, 2017.

[21] Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23(23):1–21, 2011.

[22] K. A. Kazakov and V. A. Semenov. Overview of modern motion planning methods. *Proceedings of the Institute for System Programming of the RAS*, 28(4):241–293, 2016.

[23] 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.

[24] V. V. Solovyev, I. O. Shapovalov, and V. V. Shadrina. Trajectory planning of a moving object using voronoi diagram. *Bulletin of the Southern Federal University. Technical Sciences*, (2 (163)):29–40, 2015.

[25] S. Chakravorty and S. Kumar. Generalized sampling-based motion planners. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 41(3):855–866, 2011.