

Vectorized Visibility Graph Planning with Neural Polygon Extraction

Kirill Kasmynin[1] and Konstantin Mironov[1,2,3](✉)

[1] Moscow Institute of Physics and Technology, 9 Institutsky lane, Dolgoprudny, Moscow 141701, Russia
mironovconst@gmail.com
[2] AIRI Artificial Intelligence Research Institute, 6/2 Presnenskaya Emb., Moscow, Russia
[3] Ufa University of Science and Technology, 32 Zaki Validi St., Ufa 450076, Russia

Abstract. This article proposes an approach for path planning in environments with obstacles. The methodology integrates recent neural obstacle polygon extraction with visibility graph path planning, complemented by the integration of vectorization techniques. That significantly enhances path planning efficiency. The modular design of the neural network method, encompassing contour detection, vertex identification, and polygon approximation modules, facilitates improved performance compared to traditional methods. Furthermore, the investigation into vectorization's impact on the intersection operation accelerates algorithm speed, contributing to faster path planning processes. Experimental results validate the efficacy of the integrated approach, showcasing notable improvements in path planning efficiency, especially with the utilization of vectorization techniques. The study systematically addresses challenges such as slow graph construction and inaccurate obstacle detection, providing a robust solution for optimizing path planning processes. Moreover, the implementation of multiple modules in the methodology enables its versatility for testing with various environments. This versatility allows researchers to assess the method's performance across diverse scenarios and visualize the results effectively. Overall, the integrated approach offers a comprehensive solution for optimizing path planning in complex environments, demonstrating its potential to streamline path planning processes and improve mobile robot navigation.

Keywords: Visibility Graph · Polygon Extraction · Path Planning · Neural Network · Mobile Robots · Obstacle Avoidance

1 Introduction

Path planning among the obstacles is an essential task for the autonomous motion of mobile robots. The concept of visibility graphs provides a way to solve this task [1, 2]. Search-based planning on visibility graphs offer a fast solution, however, two main challenges persist in their application. Firstly, the construction of the visibility graph itself can be time-consuming, especially in complex environments with numerous obstacles. Second, the method require representing obstacles with polygons. Obtaining such a

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
A. Ronzhin et al. (Eds.): ICR 2024, LNAI 14898, pp. 265–280, 2024.
https://doi.org/10.1007/978-3-031-71360-6_20

representation can also be computationally and time intensive. This research aims to address these challenges by integrating visibility graph with recent [3] neural method for obstacle polygon extraction from images.

In the realm of path planning, algorithms such as A*[4] and Theta* are frequently employed to navigate through environments with obstacles [5]. These algorithms offer efficient solutions by leveraging heuristic strategies in pathfinding processes. However, this research focuses on the use of visibility graphs combined with optimal polygon construction. By leveraging the capabilities of neural networks, we aim to streamline the process of extracting obstacle polygons from maps, thus reducing the computational burden. Furthermore, we propose novel algorithms to expedite the construction of visibility graphs in complex environments. By combining these approaches, we seek to develop an efficient and robust method for generating trajectories of mobile robots in environments with obstacles.

2 Related Works

Path planning is a fundamental task in mobile robot motion control, aiming to navigate from a given initial point to a specified goal point while considering potential obstacles [6]. The planning process involves defining a sequence of states for the robot to transition from its current position to the goal, with each state comprising the necessary variables to precisely determine the robot's position at time [7]. Transitioning to the goal state requires adherence to physical constraints such as time, control inputs, and obstacle avoidance. Trajectory optimality entails minimizing or maximizing quality criteria such as movement time or path length [7]. There is also a distinction between the use of planning in static and dynamic environments [8]. Here we consider planning in the static environment. Classic search strategies are based on heuristic rules specific to the given task [9]. An example of a heuristic strategy is the A* algorithm [4], which is an extension of Dijkstra's graph search algorithm [10] with the heuristic function estimating the distance from the considered point to the end point. However, algorithms based on cell decomposition have significant limitations, such as a limited number of directions caused by the grid structure. Various methods have been developed to overcome this limitation, including D* [11] and Theta* [5].

Visibility graph path planning [2] do not rely on cell representation; it require obstacles to be modeled with polygons. Image processing techniques may be used to extract polygons from the binary image. In [12] edge points are extracted from the blurred image, and the topological distribution of these points is analyzed. After extracting the edge points, the possible next step involves approximating the polygons to represent the obstacles in the environment accurately. This approximation process aims to downsize the vertices of the extracted polygons and analyze the local curvature of the obstacles based on the inner angles between connected edges [13]. An effective neural network (PolyWorld) for extracting polygons from images is described in [3]. It is designed to directly extract building vertices and connect them into polygons.

Algorithm for path planning in unmanned surface vehicles, employing a combination of quadtree representation and visibility graph techniques is presented [14]. [15] introduce an approach based on the Essential Visibility Graph for path planning and

real-time collision avoidance in multiple unmanned aerial vehicles. [16] tackled the path planning challenge for biped robots in static environments by integrating visibility graph with sampling-based planning. [17] develop an efficient shortest path finding algorithm named, utilizing visibility graphs for mobile robots navigating polygonal maps. [18] presented a dynamic free-space roadmap approach for ensuring safe motion planning of quadrotors, utilizing visibility graph concepts.

The task of trajectory planning on a plane between fixed points is one of the key problems studied in robotics [19]. The wide range of applications of developed methods, algorithms, and models in the context of solving this problem covers the field of development and design of unmanned vehicles and robotic systems capable of autonomous operation in the environment. Trajectory planning on a plane is considered as a problem of finding the shortest path on a weighted graph. In this context, the vertices of the graph represent coordinates of points on the plane, and the edges correspond to elementary trajectories, such as line segments. Elementary trajectories can are associated with characteristics such as length, complexity, and danger, which can be formalized as edge weights in the graph. The general scheme of the algorithm for constructing the shortest trajectory between two points on the plane includes two main steps: building a weighted graph and finding the shortest path on the constructed graph. In this context, the length of the constructed trajectory is considered as the main criterion of optimality [19].

In [7], graph-based methods are considered. They have various advantages, including scalability, no need for extensive preprocessing, the ability to consider kinematic and dynamic constraints, applicability in dynamic environments, and the ability to use in conditions of expanding the visibility map [7]. Algorithms based on representing the states of a moving object as a graph are widely used [20]. These include the visibility graph planning algorithm, described in the paper [2], Voronoi diagrams [21], rapidly exploring random trees [22], roadmaps [23].

Visibility graphs are a graph model applied to solve the trajectory planning problem in spaces with polygonal impassable areas. In this model, vertices correspond to polygon vertices and key points (start and end points), and graph edges connect these vertices, avoiding intersection with the inner areas of polygons [19]. The process of constructing a visibility graph is based on the known coordinates of polygon vertices, which define impassable zones. The high complexity of constructing the visibility graph is determined by the need to check for intersections with obstacles of all straight lines connecting vertices pairwise. Nevertheless, this method can be successfully applied in practical tasks [7].

3 Background

3.1 Formal Problem Statement

The problem considered in this research involves identifying obstacles in images and constructing a visibility graph for path planning. Given a set of images I, the objective is to determine the set of obstacles $O = \{O_1, O_2, \ldots, O_n\}$, where each O_i represents a polygon outlining an obstacle in the image. The output of the algorithm is the set of polygons $P = \{P_1, P_2, \ldots, P_m\}$ and a visibility graph G(V, E), where V represents the vertices corresponding to the vertices of the polygons P, and E represents the edges

connecting vertices through which movement is possible. For construction visibility graph the problem considered in this research involves finding the shortest path on that graph. In a two-dimensional Euclidean space denoted by L with a specific Cartesian coordinate system, we define a region U, representing the workspace, as $x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$. To construct the visibility graph within this region, a collection of obstacles is introduced, each represented as a convex polygon. The initial and goal points, denoted as s and g respectively, are positioned within the workspace. Furthermore, the obstacle polygons, delineated by vertices p on the map, are identified for constructing the visibility graph. The primary objective is to determine the shortest path from the initial point s to the destination point g, while considering obstacles within the workspace. This path must traverse the edges of the visibility graph, connecting the initial point s to the destination point g. The goal is to merge methods to achieve optimal obstacle identification and efficient path planning using the visibility graph.

3.2 Polygon Extraction from Images

PolyWorld [3] serves as the foundation for extracting polygons from images. PolyWorld comprises three main components:

1. Vertex Detection Network: This component identifies potential building corner candidates in the input image.
2. Attentional Graph Neural Network: Responsible for aggregating information from vertices and refining their positions using graph neural network techniques.
3. Optimal Connection Network: Generates valid connections between vertices based on a permutation matrix.

This approach provides both the positions of detected corners and a valid permutation matrix for polygon construction. In addition to the neural network-based approach, alternative methods for polygon extraction were explored. This involved developing modules for contour extraction from binary images, with and without vertex approximation. Experiments were conducted to compare the neural network-based implementation with these alternative methods. Contour extraction from binary images was accomplished using OpenCV's contour detection functions. Initially, the contours were detected using the findContours. The approxPolyDP function was subsequently applied to approximate the contours, allowing for vertex simplification based on the Douglas-Peucker algorithm [13]. Performance evaluation of the neural network-based method and alternative methods focused on the number of vertices obtained. Additionally, comparisons were made before and after vectorizing the intersection checking operation to assess its impact on overall process efficiency.

4 Methodology

The formula for finding the shortest path in the visibility graph can be represented as the problem of finding the optimal path considering visible lines and obstacle positions. This allows optimizing the trajectory of a moving object, taking into account its movement and spatial constraints [2]. Construction of the graph include the following steps:

1. Identifying obstacle polygons on the map. These polygons represent the regions where movement is restricted.
2. Utilizing the vertices p obtained from these obstacle polygons when constructing the visibility graph. These vertices define the boundary points of the obstacles.
3. Connecting each point p within the working area U, including the initial (s) and final (g) points, to form the initial graph structure.
4. For each pair of points p_1 and p_2, verifying if the line segment p_1p_2 intersects with any obstacles. If not, add this line segment to the visibility graph, indicating a direct line of sight between the points.

Planner implementation consists of several modules, each serving a specific purpose in the path planning process. Firstly, obstacles extraction module handles the loading of obstacles from module of obstacle extraction or their generation within the program. This module ensures that the obstacles are properly formatted and prepares them for integration into the visibility graph. Second, graph construction module is responsible for constructing the visibility graph based on the provided obstacles and defining the start and goal points for path planning. This module includes functions for building the visibility graph and adding the necessary vertices and edges to represent the obstacles and the start and end points. Next, the Dijkstra's algorithm module finding the shortest path on the constructed visibility graph. It includes functions for executing the Dijkstra algorithm, determining the shortest path.

4.1 Graph Construction

The algorithm for constructing a visibility graph in pseudocode is presented below.

```
Procedure ObstacleBuild(O):
    E_O = ∅
    V = ∅
    For each O_i in O:
        For j = 1, 2, ..., m_i-1:
            e = (v_{i,j}, v_{i,j+1})
```

```
                E_O = E_O ∪ {e}
                V = V ∪ {v_{i,j}, v_{i,j+1}}
    Return E_O, V

Procedure VisGraph(V, E_O):
    E_V = ∅
    V_open = V
    While V_open ≠ ∅:
        v = V_open[0]
        V_open = V_open \ {v}
        V_visited = V_visited ∪ {v}
        For each v' in V_open:
            e = (v, v')
            If ¬IsEdgeIntersecting(e, E_O):
                E_V = E_V ∪ {e}
    Return E_V

Procedure AddPoints(p_O, p_E, V, E_O):
    E_V' = ∅
    For each v in V:
        e_O = (p_O, v)
        e_E = (p_E, v)
        If ¬IsEdgeIntersecting(e_O, E_O):
            E_V' = E_V' ∪ {e_O}
        If ¬IsEdgeIntersecting(e_E, E_O):
            E_V' = E_V' ∪ {e_E}
    Return E_V'

Procedure IsEdgeIntersecting(e, E_O, O):
    For each e_O in E_O:
        If e ∩ e_O ≠ ∅:
            Return True
    line1 = LineString from e
    For each O_i in O:
        p = 0
        For each v in O_i:
            If line1 ∩ v ≠ ∅:
                p = p + 1
        If p > 1:
            Return True
    Return False

Procedure AreEdgesIntersecting(edge1, edge2):
    line1 = LineString from edge1
    line2 = LineString from edge2
   Return line1 ∩ line2 ≠ ∅
```

Here O is a set of obstacles, E_O is a set of obstacle edges, V is a set of obstacle vertices, m_i: is a number of vertices in obstacle O_i, v_i,j is a j-th vertex of obstacle O_i, e is an edge between two vertices, E_V is a set of edges of visibility graph, V_visited is a set of visited vertices, V_open is a set of open vertices to be visited, p_O is a start point, p_E is an end point, E_V′ is a set of edges added to visibility graph, e_O is an edge between start point and a vertex, e_E is an edge between end point and a vertex.

`ObstacleBuild` procedure initializes the construction process by converting obstacle data into graph edges and vertices. Given a set of obstacles O, the procedure iterates through each obstacle Oi, extracting its vertices vi,j and forming edges between consecutive vertices. The resulting sets of obstacle edges EO and vertices V are returned. `VisGraph` procedure constructs the visibility graph by iteratively connecting vertices that have line-of-sight with each other, while avoiding intersections with obstacle edges. Starting with an open set of vertices Vopen, the procedure selects a vertex v and checks its visibility to other vertices. If an unobstructed edge exists between v and another vertex, it is added to the visibility graph's set of edges EV. This process continues until all vertices are visited. `AddPoints` procedure extends the visibility graph to include additional points such as the start point pO and end point pE. For each vertex v in the set of vertices V, edges connecting pO and pE to v are created. These edges are then checked for intersection with obstacle edges using the IsEdgeIntersecting function. If no intersections occur, the edges are added to the set of visibility edges EV′.

`IsEdgeIntersecting` procedure determines whether a given edge e intersects with any obstacle edges or vertices. It is essential for ensuring that the edges in the visibility graph do not penetrate any obstacles. Initially, the procedure involves checking if the edge e intersects with any edge in the set of obstacle edges EO using the `AreEdgesIntersecting` procedure. If no intersections are found, the procedure then checks for intersections with the obstacle vertices. This involves examining edge e intersection with each vertex in the obstacles O. A counter is used to count the number of vertices intersected within each individual obstacle Oi. If the counter exceeds one, it indicates that the edge e is passing through the interior of an obstacle, thus confirming an intersection. Initially, the `IsEdgeIntersecting` procedure relied on the `AreEdgesIntersecting` procedure, which used the crosses function from the Shapely library [24, 25] for intersection detection. However, for better vectorization potential, this operation was later rewritten using a Held method for intersection checking [26, 27]. `AreEdgesIntersecting` procedure evaluates whether two edges intersect by converting them into line segments and checking for any intersection points. This procedure initially utilized the crosses function from the Shapely library and was tested so that then we could see the difference between the implementation with vectorization of this operation and without. Respectively given the need for further vectorization, the procedure was later revised to implement an alternative intersection checking method.

4.2 Check for Segment Intersection

For two segments AB and CD, where A(x_1, y_1), B(x_2, y_2), C(x_3, y_3), and D(x_4, y_4), the function crosses returns True if they intersect. Let L_1 and L_2 be the lines represented by segments AB and CD respectively. If segments AB and CD intersect, they share some but not all common points. Let the intersection point be denoted as P(x_p, y_p). Then the

intersection will be one-dimensional and have a dimension smaller than the maximum dimension of either segment AB or CD. Additionally, the intersection will not be equal to either segment AB or CD.

In mathematical terms, this can be represented as:

- There exists a point P that belongs to both segments: $P \in AB$, $P \in CD$.
- The intersection has a dimension smaller than the maximum dimension of either segment: $\dim(P) < \max(\dim(AB), \dim(CD))$.
- The intersection is not equal to either segment: $P \neq AB$, $P \neq CD$.

Based on this definition, the crosses function returns True if all these conditions are satisfied for the given segments AB and CD, and False otherwise.

To ensure efficient vectorization of the edge intersection check, Held's method is utilized, which employs the determinant to determine the side of a point relative to a segment. Held's method is a robust approach for intersection detection. Held's method forms a determinant to determine the side of a point p against a segment ab. The determinant is given by:

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

This determinant can be interpreted geometrically. Specifically, a point p is strictly to the left of the oriented line through a and b if $\det(\mathrm{a}, \mathrm{b}, \mathrm{p}) > 0$. Through simple algebraic operations, it follows that $\det(\mathrm{a}, \mathrm{b}, \mathrm{p}) = \mathrm{ab} \times \mathrm{ap}$. The conditions for segments AB and CD to intersect can be determined based on this concept, fully aligning with the conditions described in the method [26, 27]. To leverage Held's method for vectorization, calculation is replaced with a vectorized approach. Below is a description of how this implementation is achieved in the algorithm.

```
Procedure: IsEdgeIntersecting(new_edge)
    obstacle_edges = array of obstacle edges
    new_edge_coords = array of new edge coordinates

    x1 = obstacle_edges[:, 0]
    y1 = obstacle_edges[:, 1]
    x2 = obstacle_edges[:, 2]
    y2 = obstacle_edges[:, 3]

    x3 = Duplicate(x_{new_edge}[0], |obstacle_edges|)
    y3 = Duplicate(x_{new_edge}[1], |obstacle_edges|)
    x4 = Duplicate(x_{new_edge}[2], |obstacle_edges|)
    y4 = Duplicate(x_{new_edge}[3], |obstacle_edges|)

    determinant1 = (x1 * y2 + x2 * y3 + x3 * y1) - (y1 *
x2 + y2 * x3 + y3 * x1)
    determinant2 = (x1 * y2 + x2 * y4 + x4 * y1) - (y1 *
x2 + y2 * x4 + y4 * x1)
    determinant3 = (x3 * y4 + x4 * y1 + x1 * y3) - (y3 *
x4 + y4 * x1 + y1 * x3)
    determinant4 = (x3 * y4 + x4 * y2 + x2 * y3) - (y3 *
x4 + y4 * x2 + y2 * x3)

    intersects = (determinant1 * determinant2 < 0) and
(determinant3 * determinant4 < 0)

    if True in intersects:
        return True
    end if

    line1 = LineString from new_edge

    for O_i in obstacles:
        p = 0
        for v in O_i:
            if line1 ⋂ v ≠ ∅:
                p = p + 1
            end if
        if p > 1:
            return True
        end if
    end for
    return False
```

The function `IsEdgeIntersecting` is designed to check if a new edge intersects with any obstacle edges using a vectorized approach based on Held's method.

In that alghorithm the edges of obstacles are converted into a NumPy array, `obstacle_edges`, with each edge represented by the coordinates of its start and end points. The coordinates of the new edge, new_edge_coords, are also converted into a NumPy array. The coordinates of the start and end points of the obstacle edges are extracted into separate arrays x_1, y_1, x_2, y_2. The coordinates of the new edge are duplicated to match the number of obstacle edges. This ensures that each element in the new edge array can be compared with the corresponding element in the obstacle edge array.

Determinants are computed for the cross products required by Held's method. They used to check if the new edge intersects any of the obstacle edges. By duplicating the new edge coordinates to match the number of obstacle edges, the algorithm can leverage vectorized operations for efficient computation, significantly speeding up the intersection check process new edge with all obstacle's edges at the same time. If any element in the intersects vector is True, it indicates that there is an intersection, and hence the edge is not added to the visibility graph.

5 Experiments and Results

The purpose of the experiments is to evaluate the performance and efficiency of the proposed trajectory planning method based on the visibility graph.

Initially, the visibility graph construction algorithm was executed using randomly generated polygonal regions, with the quantity thereof varying. Subsequently, the pathfinding methodology, reliant upon the visibility graph and employing Dijkstra's algorithm, was applied. Building the visibility graph when generating various polygons for 13 obstacles takes about 4 s. Pathfinding on the already constructed graph takes around 1 ms. Since the pathfinding speed is quite fast, using an algorithm other than Dijkstra's does not make sense. The constructed graph with generated obstacles is shown in Fig. 1.

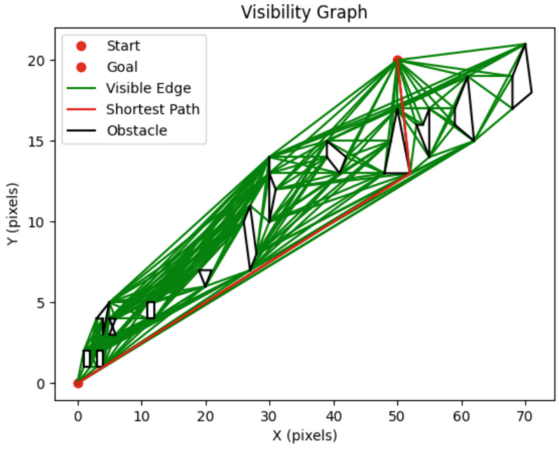

Fig. 1. Visibility graph with generated obstacles

After integrating visibility graph with PolyWorld we proceeded to extract polygons from images. Example image contained 10 obstacles and 85 vertices included start and goal vertices. Pathfinding using the Dijkstra algorithm took approximately 1–1.5 ms, while constructing the visibility graph took about 10.2 s. Obtaining information about obstacles during the execution of the module for obtaining polygons took approximately 3 s per image to extract obstacle polygons and obtain them as a list of vertices. Then the obtained result is displayed at Fig. 2. The experiments demonstrate the effectiveness of the proposed visibility graph-based trajectory planning method, particularly in environments with multiple obstacles. These experiments were conducted without vectorization to evaluate different methods of polygon extraction and their impact on computational efficiency and accuracy. Subsequently, the process was analyzed with vectorization to compare the performance improvements.

Initially, a binary image was generated from the segmentation data. A method was developed to extract obstacle vertices from the binary image. Without employing approximations, the vertices were accurately identified using the `cv2.findContours()` function, which detects contours on the binary image and places points accordingly. These vertices were saved and processed to transform them into suitable vertices for constructing the visibility graph. In that case process of visibility graph construction took approximately 170–200 s, resulting in a graph comprising 208 vertices, including start and goal points, and 11 obstacles. The result is shown in the Fig. 3.

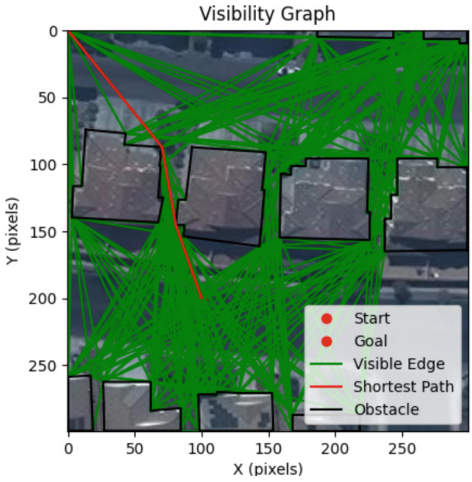

Fig. 2. Visibility graph and PolyWorld

In the case neural network implementation handled 10 obstacles and 85 vertices, including start and goal points, with graph construction taking approximately 10.2 s. It is noteworthy that the neural network approach did not rely on binary image input. The result is shown in Fig. 3. In next method of extraction with the utilization of approximation techniques, the extraction of obstacle vertices from the binary image was refined. The `cv2.findContours()` function was used to detect contours, and

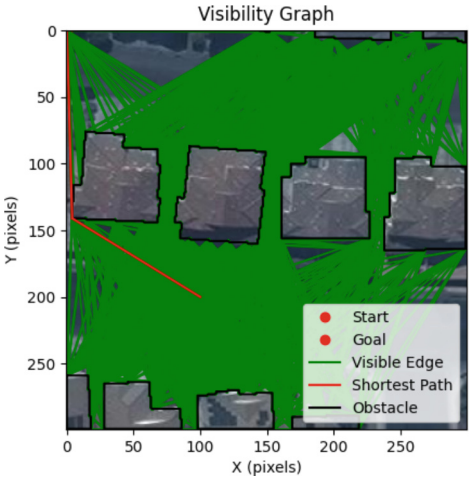

Fig. 3. Visibility graph constructed without approximations

the `cv2.approxPolyDP()` function was employed to approximate each detected contour. The approximated vertices were saved and processed for visibility graph construction, resulting in a graph that was built in approximately 6.4 s. This graph included 69 vertices, including start and goal points, and 11 obstacles. However, some parts of the buildings were not accurately represented within the obstacles. The result is shown in the Fig. 4.

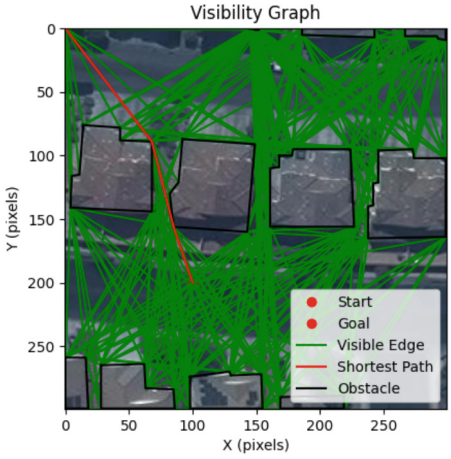

Fig. 4. Visibility graph constructed with approximations

The experiments aimed to determine the effectiveness of different polygon extraction approaches and their suitability for trajectory planning applications. Results were

analyzed to identify the most efficient method based on performance metrics such as speed of graph construction based on number of vertices and vertex accuracy.

Based on the conducted comparison, it can be concluded that the neural network implementation demonstrates a relatively small number of vertices and relatively fast graph construction time. Therefore, the neural network implementation may be the preferred choice for similar tasks, providing a balance between accuracy and efficiency.

The intersection operation is vectorized utilizing the algorithm detailed earlier. Consequently, its application alongside PolyWorld yields an identical graph to that obtained without vectorization (see Fig. 2), but with significantly accelerated construction speed on the same image, reducing to 2.32 s from 10.2 s.

When an experiment is conducted involving 13 generated obstacles, the resultant graph remains unchanged from the previous iteration. However, the construction speed is notably enhanced, taking only 0.6 s instead of the prior 4 s. Upon employing the visibility graph construction on vertices obtained through non-neural network methods from binary images, a construction speed of 1.53 s is achieved when approximating the obstacles, and 36 s without approximation. The presented results (Table 1) confirm a significant acceleration in graph construction process through vectorization.

We also conducted experiments with Theta* and A* using the same obstacles as with the visibility graph. Initially, the maps were represented differently to standardize the environment representation, we converted binary images into the occupancy grid format for implementation Theta* and A*. This allowed us to compare the visibility graph with other algorithms on the same map, noting that the environment representations differ – one being a list of obstacle vertices from a 300×300 image, the other an occupancy grid of the same size. We executed the Theta* and A* algorithms and measured the environment construction time from the binary image, which took approximately 86 ms, whereas using Poly World takes 2.5 s. It is important to note that in this case, the segmentation task is not addressed, and a binary image is provided directly, unlike the PolyWorld neural network approach which requires an RGB input. The pathfinding time for Theta* is about 6.286 s, considering both the environment construction and pathfinding phases. These results are presented in Table 1.

Table 1 compares various methods for environment construction and path planning. Experiment 1 uses PolyWorld with visibility graph and Dijkstra's algorithm. Experiments 2 and 3 test non-neural network methods without and with approximation, respectively, showing longer construction times for the first. Experiment 4 integrates vectorized intersection with PolyWorld and visibility Graph, highlighting the benefits of vectorization in speeding up the process. Experiments 5 and 6 with and without approximation demonstrate faster execution times but lower accuracy for the first. Experiments 7 and 8 evaluate Theta* and A* algorithms, providing benchmarks. Experiment with vectorized intersection and PolyWorld (Experiment 4) efficient compared to traditional non-neural network approaches (Experiments 2 and 3) and competitive with Theta* and A*. In future work, real-world testing will be conducted to supplement the simulated results and validate the method's performance under practical conditions.

In conclusion, the integration of vectorization profoundly expedites the graph construction process, while the neural network method retains its superiority in terms of both

Table 1. Environment Construction and Pathfinding Time

Experiment	Method	Execution time
1	PolyWorld + Visibility Graph + Dijkstra	12.7 s
2	Non-Neural Network (No Approximation) + Visibility Graph + Dijkstra	170–200 s
3	Non-Neural Network (With Approximation) + Visibility Graph + Dijkstra	6.4 s
4	Vectorized Intersection + PolyWorld + Visibility Graph + Dijkstra	4.8 s
5	Vectorized Intersection with Non-Neural Network (with approximation) + Visibility Graph + Dijkstra	1.53 s
6	Vectorized Intersection with Non-Neural Network (without approximation) + Visibility Graph + Dijkstra	36.05 s
7	Theta*	6.286 s
8	A*	50.1 s

accuracy and speed. Hence, the amalgamation of vectorized visibility graph construction and PolyWorld emerges as an efficacious approach in path planning.

6 Conclusion

In this work we explore the possibility of acceleration the visibility graph path planning via neural obstacle decomposition and vectorization of the graph construction procedure. Experiments have shown that the algorithm successfully finds the shortest path on various sets of obstacles. Through a series of experiments, the proposed trajectory planning method's performance and efficiency were evaluated, highlighting its effectiveness. Notably, the integration of a neural network-based obstacle detection method significantly expedites the obstacle extraction process, streamlining the generation of obstacle polygons. Furthermore, the experiments demonstrate the impact of algorithmic optimizations, data structure enhancements, and vectorization on the trajectory planning system's overall efficiency. Additionally, the integration of polygon extraction functionalities enhances the trajectory planning system's adaptability and robustness, allowing it to handle diverse environments with ease. A promising direction of future research consist in integration with polygon-based approaches [28] on local collision avoidance with model-predictive control.

References

1. Yang, F., Cao, C., Zhu, H., Oh, J., Zhang, J.: FAR planner: fast, attemptable route planner using dynamic visibility update. arXiv:2110.09460 (2021)
2. Lozano-Pérez, T., Wesley, M.: An algorithm for planning collision-free paths among polyhedral obstacles. Commun. ACM **22**(10), 560–570 (1979)

3. Zorzi, S., Bazrafkan, S., Habenschuss, S., Fraundorfer, F.: PolyWorld: polygonal building extraction with graph neural networks in satellite images. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 1848–1857 (2022)
4. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
5. Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: any-angle path planning on grids. J. Artif. Intell. Res. **39**, 533–579 (2010)
6. Liu, V.: Path planning methods in obstacle environment (review). Math. Math. Model. 15–58 (2018)
7. Pshikhov, V., Medvedev, M., Brosalin, D., Vasilieva, M., Gurenko, B., Hamdan, N.: Research on motion planning methods in two-dimensional mapped environments. Bull. South. Fed. Univ. Tech. Sci. **3**, 170–192 (2022). (in Russian)
8. Afanasov, A.: Analysis of path planning methods for autonomous mobile devices. Issues Sci. Educ. **17**(64), 9–17 (2019). (in Russian)
9. Yahja, A., Stentz, A., Singh, S., Brumitt, B.: Framed-quadtree path planning for mobile robots operating in sparse environments. In: Proceedings of the IEEE International Conference on Robotics and Automation 1998, Leuven, Belgium, pp. 650–655. IEEE (1998)
10. Dijkstra, E.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
11. Sadiq, A., Raheem, F., Abbas, N.: Optimal trajectory planning of 2-DOF robot arm using the integration of PSO based on D* algorithm and cubic polynomial equation. In: The First International Conference on Engineering Researches, pp. 458–467 (2017)
12. Suzuki, S., Be, K.: Topological structural analysis of digitized binary images by border following. Comput. Vis. Graph. Image Process. **30**(1), 32–46 (1985)
13. Douglas, D., Peucker, T.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: Int. J. Geogr. Inf. Geovisualization **10**(2), 112–122 (1973)
14. Lee, W., Choi, G.-H., Kim, T.-W.: Visibility graph-based path-planning algorithm with quadtree representation. Appl. Ocean Res. **117**, 102887 (2021)
15. Blasi, L., D'Amato, E., Mattei, M., Notaro, I.: Path planning and real-time collision avoidance based on the essential visibility graph. Appl. Sci. **10**(16), 5613 (2020)
16. Pradhan, S., Mandava, R., Vundavilli, P.: Development of path planning algorithm for biped robot using combined multi-point RRT and visibility graph. Int. J. Inf. Technol. **13**(4), 1513–1519 (2021)
17. Missura, M., Lee, D., Bennewitz, M.: Minimal construct: efficient shortest path finding for mobile robots in polygonal maps. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, pp. 7918–7923. IEEE (2018)
18. Guo, J., Xun, Z., Geng, S., Lin, Y., Xu, C., Gao, F.: Dynamic free-space roadmap for safe quadrotor motion planning. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2022, Engineering, Computer Science, Kyoto, Japan, pp. 10523–10528. IEEE (2022)
19. Yuan, X., Shi, J., Gu, L.: A review of deep learning methods for semantic segmentation of remote sensing imagery. Expert Syst. Appl. **169**, 114417 (2021)
20. Kazakov, K., Semenov, V.: Overview of modern motion planning methods. Proc. Inst. Syst. Program. RAS **28**(4), 241–294 (2016)
21. Solovyev, V., Shapovalov, I., Shadrina, V.: Trajectory planning of a moving object using Voronoi diagram. Bull. South. Fed. Univ. Tech. Sci. **2**(163), 29–40 (2015). (in Russian)
22. Chakravorty, S., Kumar, S.: Generalized sampling-based motion planners. IEEE Trans. Syst. Man Cybern. Part B (Cybern.) **41**(3), 855–866 (2011)

23. Kavraki, L., Svestka, P., Latombe, J., Overmars, M.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robot. Autom. **12**(4), 566–580 (1996)
24. Shapely: A Package for Manipulation and Analysis of Geometric Objects (2024)
25. GEOS coordinate transformation software library. Open Source Geospatial Foundation. https://libgeos.org/. Accessed 25 May 2024
26. Held, M.: ERIT – a collection of efficient and reliable intersection tests. J. Graph. Tools **2**(4), 25–44 (1997)
27. Jokanovic, S.: Two-dimensional line segment-triangle intersection test: revision and enhancement. Vis. Comput. **35**, 1347–1359 (2019)
28. Logunov, A., Alhaddad, M., Mironov, K., Yakovlev, K., Panov, A.: Polygon decomposition for obstacle representation in MPC-based motion planning. J. Intell. Robot. Syst. (under review)