Planar Waypoint Generation and Path Finding in Dynamic Environment

Daoyuan Jia[1], Cheng Hu[1], Kechen Qin[1], Xiaohui Cui[1]*
[1]International School of Software, Wuhan University
Wuhan, Hubei, P.R.C
Email: xcui@whu.edu.cn

***Abstract*—Path Planning stands in basic positions in robotics, game AI and navigation. Previous solutions generally focused on decompose the environment into grid-like map, which is large in proportion to the size and complexity of the environment, and thus the efficiency of graph construction, update and path-finding suffers. We proposed a waypoint generation method to discretize the environment into a waypoint graph. By utilizing waypoint filtering and edge sparsification, we control the size of waypoint graph to be relatively small without sacrifice path quality in path finding. Our method limits the length of edge, and thus supports fast local update for dynamic environment. Experiments show that our way can fast generate the waypoint graph from continuous environment, and update it locally and dynamically. By integrating with physics engine, we also support path finding for multiple agents in a dynamic environment.**

Keywords-waypoint;waypoint graph generation; path finding

I. INTRODUCTION

Path Planning stands in basic positions in robotics, game AI and navigation. Mostly the whole process can be divided into two distinct steps: Firstly discretize the continuous environment and represent it into a queryable structure, generally graph structure; secondly, do the path-finding from a given location within the map. The second step, path planning over waypoint graph, can be regarded as a simple path-find problem on a graph[1][2].

Waypoint graph is a common solution to discretize and represent the environment especially in Computer Graphic applications [3]. It guarantees that a character of specific size can walk through each connected edge in waypoint graph without making any collision to obstacles. But waypoints are open manual constructed by developers and designers, which requires experience and is tedious for most of people especially when the map is large and complicated. We proposed a way to, for any given character size, automatically generate a locally updatable waypoint graph from input map. By checking trace area rather than trace line, our approach makes each edge in waypoint graph is strictly traversable for specific-sized characters, which makes waypoint graph even more practical for simulation or robot navigation.

Our waypoint graph can be categorized in to corner graph [5]. In construction process, we firstly generate and filter candidate waypoints based on corner, and then use a post process to exclude unnecessary vertices and carefully sparsify the edges without sacrifice connectivity. This method will control the total vertices and edges of the generated graph to a more reasonable degree in environment representation step comparing with other methods. After that, we segment long edges to limited length, which facilitates quick waypoint graph update only a part of the map. Our experiments show that the post process neither brings much overhead in construction nor notably increases the path length comparing with shortest path in path-finding step.

II. RELATED WORKS

As predecessors' work of this field these years, many methods have been introduced about the discretization and representation. They can be roughly divided into grid based solutions, which discretize the environment into equal sized squares, and non-grid based ones.

For grid based solutions, potential field method, which is firstly introduced by Andrews and Hogan[6], discretizes the continuous environment into grids with different values based on obstacle’s positions, and the grid map is called potential field. As the grid map refreshing in every time step, potential filed solution is suitable for dynamic environment and multi-agents interaction[8]. Borenstein and Koren[7] which combined potential filed with a path finding algorithm, and is able to be used in environment with highly complicated obstacles. However, the complicity is directly related to the number of grids, which is determined by the character’s size and the minimum size of exit in the environment. If the exit size is too small, the grid size must be small enough to obtain correct path finding solution.

As for non-grid based solutions, to name few, Lozano-Pérez et al.[3] introduced visibility graph, which can be regarded as an early form of waypoint graph generation method. This method inflates the obstacles for characters of specific shape, and use corner points as the waypoints. Pettre et al.[4] proposed navigation graph with a little modification on visibility graph, which assists different-sized characters on multi-layered surface. However, the common drawback of these methods is fail to support dynamic environment. Wardhana et al. [14] proposed a waypoint graph generation method in 3D environment and specified character’s movement as surface and volumetric move. Also, they utilized methods to reduce the waypoint and edges to boost path finding process without bring poor quality to path finding result[15]. These methods tend to regard the environment as static, and the changes in the environment may cause big overhead when updating the whole graph. Moreover, these methods do not consider the multi-character situation.

III. Waypoint Graph Generation

A. Input and Output

We require three types of input. The first type is the map data, which is represented with shapes, closure ones like polygons, or non-closure ones like segment lines. Shapes are constructed with a serial of two dimensional vectors. Tags are also attached to each shape, which to specify the shape to be the map border, close or open, hollow obstacle or solid one. Such kind of planar map structure is easy to parse and commonly seen in GIS applications. Another input required is a list of character's sizes or shapes. We simplify the process by consider all characters to be a circle. For a polygon-shaped character, we can calculate it's outside Minimum Bounding Circle with Emo Welzl's algorithm[11] and get the radius of it.

The output is a non-direction graph with each vertex to be a waypoint of the map and each edge to be weighted by length. Waypoints represent important locations in the map like corners of obstacle. Waypoints are not directly the corner vertices but deviate from the corner a little bit since they may be too closed to obstacle and thus not reachable. The offset is bind to certain character size.

B. Corner Waypoint Generation

Our way of waypoint generation contains mainly two steps. Firstly potential waypoints are generated and collected according to corners of obstacles and map boundary; we call them as waypoint candidates. Secondly the waypoint candidates are filtered by some principles.

1) Waypoint Candidates Determination: There are several types of obstacle to be concerned with. Classified by purpose, they can be divided into boundary type and non-boundary type; distinguished by shape, they can be classified as closed solid obstacle (polygon), closed hollow obstacle (polygon) and open obstacle (segment). They are all labeled with tags. Boundary is optional for an environment. If exist, it must be a closed hollow polygon and all other obstacle must not exceed it. Non-boundary open obstacle is composed with line segments, which can used to simplify the representation of the element in the environment such as handrail. Non-boundary closed obstacles are used to represent normal solid hindrances such as pillar, or hollow region, such as a room temporarily sealed. All will be treated with method almost the same.

Any closed polygon shape convex or concave, with a series of vertices $V_1, V_2, V_3 \dots V_n;\ n \geq 3$, can be denoted by edges $\overrightarrow{V_1V_2}, \overrightarrow{V_2V_3}, \dots \overrightarrow{V_{n-1}V_1}$. For corner θ_{n-1} composed by $\overrightarrow{V_{n-2}V_{n-1}}$, $\overrightarrow{V_{n-1}V_n}$, and character with radius r, we can identify a vector $\vec{v}$

$$\vec{v} = \frac{\overrightarrow{V_{n-2}V_{n-1}}}{\left|\overrightarrow{V_{n-2}V_{n-1}}\right|} + \frac{\overrightarrow{V_nV_{n-1}}}{\left|\overrightarrow{V_nV_{n-1}}\right|} \tag{1}$$

Thus we can define a location $\vec{w}$ as waypoint candidate that satisfy:

$$\vec{w} = \overrightarrow{v_{n-1}} + \frac{\vec{v}}{|\vec{v}|} \tag{2}$$

$\overrightarrow{v_{n-1}}$ is the location of vertex V_{n-1}. $\vec{w}$ is either in or out of the obstacle, and on the angular bisector of the corner. We can conclude that for each pair of adjacent corners the waypoint candidates of both outer pair, if not hindered by other obstacle, are reachable, so does the inner pair.

2) Candidates Waypoint Validation and Cluster: After previous process, we can get a bunch of waypoint candidates which many of them are invalid or unnecessary and should be removed or replaced for better performance in later process. We define a candidate waypoint as validate when it is not being hindered by other obstacle and within the boundary. Fig. 1(a) has shown typical situations that a candidate waypoint get hindered by other obstacle. Hindered means other obstacle intersect with the circle that w stand on the center and radius to be r. For simplicity in implementation, we can use octagon to represent the circle. Fig. 1(b) shows an example of valid and invalid candidate waypoints.

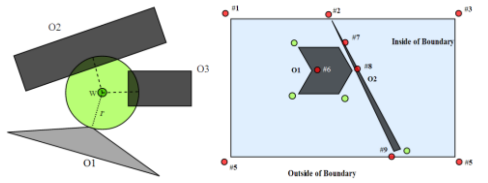


Figure 1. (a) a typical situation of an invalid candidate waypoint. O1 to O4 are obstacles, and w is a candidate waypoint generated by O1. *w* is hindered by obstacle O2, O3 and O4 as it get to close to them.(b) Example of valid and invalid candidate waypoints. Green ones are valid, while red ones are to be removed as they are either outside of or on the boundary (#1 to #5 and #9), truncated by or inside of other obstacle (#6 to #8).

We can remove candidate waypoints that are out of boundary or inside solid obstacles by testing whether the candidate is on the edge or in a closure polygon; then for each candidate waypoint W, and corresponding vertex Vn, we denote location W', where $\overrightarrow{V_nW'} = 2 \cdot \overrightarrow{V_nW}$. If there is any other edge intersect with segment VnW', W will be removed since it is hindered and not traversable for character with radius bigger than r.

Finally, we cluster candidate waypoints roughly aggregating together. Here, we discretize the map into grids for quickly querying candidate waypoints. Grids are $2r \times 2r$ sized and candidate waypoints that belong to same grid are grouped together. By examining each group, we use their geometric center as a new waypoint candidate, which serves as a replacement for the group. We recheck to make sure each waypoint candidate in the group is traversable for the replacement. If not, the waypoint will not be removed, since hindrance exists, and vice versa.

C. Reachability Test

The character's movement on the map is actually an area rather than line. Using line intersection test may discover most of collision along the edge, except some special situations. As shown in Fig. 2, character with radius r move from position green to red and there three obstacle in the way. Using the centerline to do line intersection test can only discover O2, while O1 and O3 are ignored.

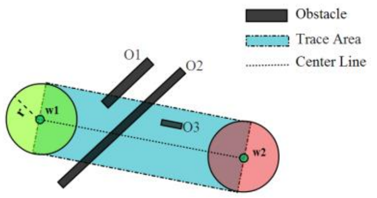


Figure. 2: All possible unreachable situations for a character moving from waypoint w1 to w2.

So the reachability should be tested with all near obstacles that whether any vertex of them in the trace area, a rectangle, or whether any line of the obstacle intersect with the center line of the trace. This process can be boosted by comparing AABB intersection of trace area and all other obstacles in order to find all near obstacles.

D. Connection Establishment

In this part, we will build waypoint graph by connecting waypoints with their neighbors. For a better performance in path planning and waypoint graph update, we applied post process to limit the maximum length of each connection and to sparsify the graph without lose connectivity.

1) Construct Complete Traversability Graph: To change the default, adjust the template as follows. To build a sparse graph, we firstly build a complete traversable graph by connect each waypoint with all other reachable waypoints use the method we introduced in Sect.3.3.

As this procedure generating numerous edges, let $|W|$ to be the number of the waypoints and $|E|$ be the number of edges, for the worse case, there will be $|E| = C^2_{|W|}$ edges, the path planning process will be slow down. So the waypoint graph should be sparsified with post processes, in which every waypoint is only connected to few other waypoints while the waypoint graph remains connectivity.

2) Incremental Sparsification: Clearly, a complete connective graph can be used to find a shortest path out of two given location; any method applied to sparsify the graph may result in loss of shortest path and make cost increase for traveling two place. Meanwhile, any connective graph can be sparsified into a Minimal Traversable Tree which still remains connectivity but may make path planning extremely costly. The balance is to sparsify some edges with probable extra overhead in path planning. We altered the method proposed by Wardhana et al[14][15], which can be grouped into θ-graph approach[14], to sparsify the dense waypoint graph.

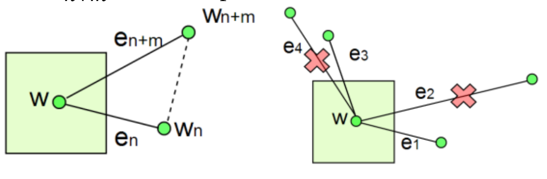
We use a square to filter closest edges of every waypoint without losing their relatively regular distribution shown in Fig. 3. Assuming we currently filter the edges from a waypoint w, we construct a square and put w in the center of it. Then we group the edges based on the square's edge ($e_{1_{sqr}}$to $e_{4_{sqr}}$) they penetrate. For each square edge $e_{n_{sqr}}$ and the edge group, we firstly remain the shortest edge from the group, and iterate from the second shortest edge to check if it should be remained. Assign e_n and e_{n+m} to be nth and $n+m$th shortest edge of the group, and w_n, w_{n+m} are the other end of them. When checking e_{n+m}, we use A* algorithm to generate the path $p_{n,n+m}$ from w_n to w_{n+m}. If the path $p_{n,n+m}+ e_n$ is not significantly longer than e_{n+m}, and total direction change during the path, denoted as θ_t is small, the path can be an good alternative for e_{n+m}, which means for a defined threshold Δ, if

$$\frac{|P_{n,n+m}| + |e_n|}{|e_{n+m}|} \times \zeta \leq \Delta \tag{3}$$

where

$$\zeta = \frac{log(m+1)}{log(\theta_t + 2 \cdot 5)} \tag{4}$$

e_{n+m} will be removed because we can use a little more effort to reach w_{n+m} through $w \to w_n \to \cdots \to w_{n+m}$; otherwise e_{n+m} will be kept.



(a) (b)

Figure 3.(a) Illustrations for incremental sparsification technique. w is the waypoint, $w_n$and w_{n+m} are reachable to w. If cost for path $w \to w_n \to \cdots \to w_{n+m}$ is not significantly longer than e_{n+m}, e_{n+m} is not that critical and can be removed. (b) An alternative minimal way to sparsify the graph. e1 and e3 are shortest edge on right and top side of the square and thus remained, e2 and e4 are not, so they are removed.

This process will be applied to each waypoint to minimize edges of each direction. Then, many edges are trimmed while remaining good traversability. As we take direction changes into consideration as well, straight and direct long edges were left. This is exactly what we need as it will remain the shortest paths as many as possible, and the path are more nature because less rigid direction change will be seen. Together with later processes, we call them as balanced way of generate waypoint graph. If graph is very dense at first, we can also use hexagon instead. Alternatively, a fast way to sparse edges is simply to remove all other edges on each side of the square except for the shortest one. Since there is no A* path finding operation and guarantee the graph is not as sparse as a tree. We regard it as Minimal Method for the waypoint graph generation.

3) Edge Interpolation: Since we already get a connective waypoint graph, it is efficient to do a one-time path-find under the condition that the environment is static and no other change in map, which is quite strict for most of the cases. When changes happen in the environment and path re-planning cannot be avoided, reapply all process before to get a new updated waypoint graph, which will be low efficacy and for real-time simulation terrible delay will be seen. We proposed a method to segment the edges and thus limit update only to a certain part of the graph. We do the segmentation by iterating each edge in waypoint graph, and if edge e's length bigger than preset value l_{max}, we cut the edge by inserting new waypoints in it. After that, edge e will become several part e_1, e_2, … , e_n with $n - 1$ new waypoints inserted and each e_i is less than l_{max}.

IV. Waypoint Graph Update

For a dynamic scene, objects in the scene may constantly be moving or rotating or sudden appearing or disappearing. All of the changes can be composited by the combinations of removing an obstacle and inserting one.

Insertion or removal an obstacle in the scene can be divided into two steps: calculate the dirty area, which is a rectangle contains the changed obstacle, and extract all the waypoints in that; rebuild connection over those waypoints.

A. *Obstacle Insertion*

Firstly we calculate the dirty area. When a new obstacle S_{new} with m vertices added in the environment at runtime, whether it is open or closed, hollow or solid, we can generating an outer enclosure rectangle $R_{enclosure}$, which guarantee distance between each point p on the $R_{enclosure}$ and each waypoint q on the S_{new} is more or equal than l_{max}.

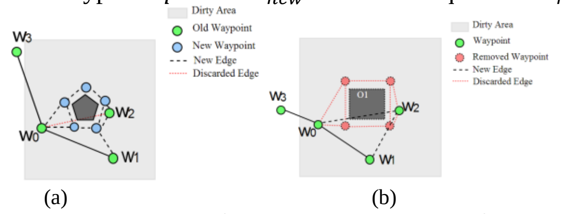


Figure 4: (a) Illustration of our way to dynamic update after inserting an obstacle. Gray pentagon is the new obstacle inserted; shallow gray area is dirty area to process; after update, there will be 5 new waypoints and 10 new edges inserted into graph, and an old edge discarded since it was hindered. (b) Illustration of our way to dynamic update after removing an obstacle. Grey square represents the removed obstacle O1. After update, there will be 4 waypoints and 8 edges to discard; and 2 edges will be added

Then for each waypoint in the $R_{enclosure}$, we discard all their edges and, as do the following operations in term: generate candidate waypoint from obstacle (Sect. 3.2.1) and validated ones are inserted into this area(Sect. 3.2.2); build complete connection in this area (Sect. 3.4.1); sparsify waypoints' edges in this area(Sect. 3.4.2) and interpolate long edges (Sect. 3.4.3). As shown in Fig. 4(a).

B. *Obstacle Removal*

Processes are similar and simpler for removing an obstacle S_{old} with k vertices from an environment. We can also create an enclosure rectangle $R_{enclosure}$ from the obstacle to be removed as dirty area. Then for each vertex v_i, $0 < i \leq k$ from S_{old}, remove its corresponding waypoint w_i (if any), and all w_i 's edges. Finally, reconstruct the connection in the $S_{enclosure}$ by connect every pair of remaining waypoint. The process is shown in Fig.4(b).

V. Path Finding

Path planning can be divided into two steps here. Firstly, insert start point and goal point into the previous generated waypoint graph by building connection with near waypoints. The nearest waypoint finding can be regarded as a Nearest N Neighbors Search problem and there are many solutions for it. We can also reuse the grid to group the waypoints for quick query. We maintain both waypoints and obstacle vertices after the waypoint candidates being filtered and clustered, which can also facilitate the waypoints' connection building and reachability test as we introduced in Sect.3.3. Nearest waypoints should pass the reachability test before connecting with start or goal point. After connection, new edges are segmented if they exceed the length limit l_{max}.

Secondly, do the path finding on the graph. There are many algorithms such like Dijstra [12], Floyd, A* [13] and many variants like Theta* [9] and D* [10]. Dijstra can be regarded as a na ïve A* without heuristic function and can be used to calculate shortest path from every waypoint to goal point on any kind of non-directional graph, while A* algorithms and the alike are used to calculate shortest path from one point to goal point with heuristic and much more efficient, but the reason for general work on grid based graph is that some heuristic functions and some other features work on grid based graph.

VI. Experiments and Results

A. *Waypoint Graph Generation performances*

We have built a 1:1 planar map of first floor of Jiedaokou Metro Station (Fig. 5a) in Wuhan of P.R.C and use it as the input map, and set waypoints' radius as equally 0.2 meter, the sparsification parameter Δ to be 16, and the edge length limit l_{max} to be 10. We compared Brutal Waypoint Generation method (Fig. 5b, 5c, 5d), which does not use post process like incremental edge sparsification and long edge interpolation, with two kinds of waypoint generation methods, both of which use post processes but have a little different in edge sparsification step. The result for initializing the waypoint graph is shown in table 1.

1) Waypoint Graph Generation Performance: The average update time for inserting and removing one obstacle from map with different waypoint generate method. The size and processing time of 3 waypoint generation methods. From left to right are Balanced Waypoint generation method, Minimal waypoint generation and Brutal waypoint generation method. CG = Candidate Points Generation, CV = Candidates Validation, CCG = Complete Connection Generation, ES = Edges Sparsification, LEI = Long edge interpolation. The hyphen means those steps are not applied.

TABLE I. Waypoint Graph Generation Performance

	Balanced	**Minimal**	**Brutal**
	#Number		
Edges	486	331	2309
Waypoints	339	266	185
	#Times (ms)		
CPG	33	32	39
CV	1051	1053	1066
CCG	353	360	344
ES	114	127	–
LEI	26	8	–
Total	1577	1580	1449

We can see from the table 1 that, the Edge Sparsification and Long Edge Interpolation will cause overhead, but is relatively small comparing to the whole building process.

Comparing with the waypoint graph generated by Brutal way, ones generated by Balanced and Minimal method contains less than 20% of edges, which can boost path planning operation and decrease memory usage. Also, reduced edges will cause less interpolation operation and there will be fewer points to be inserted after Long Edge Interpolation. Admittedly, the Long Edge Interpolation process has still brought many new waypoints and almost double the points in graph compare to Brutal way.

B. *Waypoint Graph Update Performance*

We initialized 5 tests by choosing 5 distinctly different places on the map, and dynamically add an obstacle then remove it. The later-added obstacle is designed to be a random convex polygon with constant-sized AABB. Each test is done after the waypoint graph construction finished and repeated several times to measure the average insert/remove time. The results are shown in table 2. We can see that the Minimal WPG takes least time mainly because the update process uses less time than Balanced WPG. Brutal WPG does not control the length of the edges in waypoint graph thus it cannot assume the size of affected area, and it simply refresh the whole graph rather than locally update which takes much more time comparing to other two methods. An example of dynamic update map is shown in Fig. 5e, 5f and 5g.

From the table we also can see that the average update time for obstacle insertion and removal operation takes equally time for specific method. We also notice that for Balanced WPG and Minimal WPG, when insert/remove obstacle at different place, the update time may vary.

1) Update Methods: The average update time for inserting and removing one obstacle from map with different waypoint generate method.

TABLE II. UPDATE TIME UNDER DIFFERENT METHODS

Method	**Balanced**	**Minimal**	**Brutal**
		#Insertion Time (ms)	
DAC	3.12	3.04	-
Update	22.20	15.00	322.88
		#Removal Time (ms)	
DAC	2.72	2.40	-
Update	24.92	13.96	306.16

DAC = Dirty Area Calculate

2) Update Different Positions: The average update time for inserting and removing one obstacle from different positions, in map with different waypoint generate method. Positions #P1, #P2… #P5 are shown in Fig 5.

TABLE III. TIME FOR UPDATE DIFFERENT POSITIONS

Position	**#P1**	**#P2**	**#P3**	**#P4**	**#P5**
			#Update Time (ms)		
Balanced WPG	25.70	17.70	25.90	20.60	27.90
Minimal WPG	17.40	14.60	16.20	9.20	15.00
Brutal WPG	291.60	324.60	302.10	319.80	334.50

C. *Path Panning Performance and Quality*

We uses A* algorithm with Euclid heuristic to plan the path and compare the path planning time and path quality before and after obstacle altered in the map. The path quality is measured by comparing path length with the theoretical shortest length, which is calculated by using Dijstra on graph generated by Brutal WPG. As the waypoint graph is relatively small, single path planning takes little time, we repeat the A* path finding for 100 times and assign to 4 pairs of different start and end places each time to measure average path finding time, that means total 400 times of A* path finding operation. We also examine average the path length ratio after waypoint graph being built and new obstacle inserted. The Path Length Ratio is measured by calculated path length, found by A* on graph generated by Balanced, Minimal and Brutal WPG method, comparing to the shortest path length, which is calculated by Dijstra on graph generated by Brutal WPG, from specific start to point. The results are shown in Table 4.

1) Path-Finding Time and Length Ratio: The first row shows the average path planning time for 400 times A* path finding operation. The rest three rows are path length ratio generated by different waypoint generation method. Average PLR are average path length ratio before and after new obstacle insertion.

TABLE IV. PATH FINDING TIME AND LENGTH RATIO

	Balanced	**Minimal**	**Brutal**
PP400	175.50*ms*	101.00*ms*	386.40*ms*
PLR before insertion	1.0484	1.1014	1.0029
PLR after insertion	1.0247	1.0782	1.0006
Average PLR	1.0366	1.0898	1.0017

PP400=Average path planning time for 400 A* path finding operation; PLR = for Path Length Ratio

The result shows that the average path planning time on the graph generated by balanced and Minimal method are much less than Brutal way's. Path planning takes more time for graphs by having more points and more edges. Although Brutal way contains least waypoint in the waypoint graph, the average degree of their points is much higher. As for Balanced and Minimal methods, Long edge interpolation (Sect. 3.4.3) may create many new waypoints and makes the graph contains more point, but it will not bring new edges to the graph, plus the incremental sparsification process (Sect. 3.4.2) replaces bunch of edges with few ones that have similar cost.

Brutal WPG contains completed connection and the majority of candidate points, so the A* path finding result is almost equal to the result calculated by Dijstra for specific pair of start and end point. Minimal WPG has trim edges as many as possible to keep each point has one edge in no more than one direction (that is, for each point, 4 degree at most), which may remove many essential edges, and the result shows that it brings longer path length, about 8% longer than shortest path. Balanced WPG carefully examines which edge to remove, while also wants to maintain edges as less as possible, which result in less than 4% trades off in path length comparing to shortest ones.

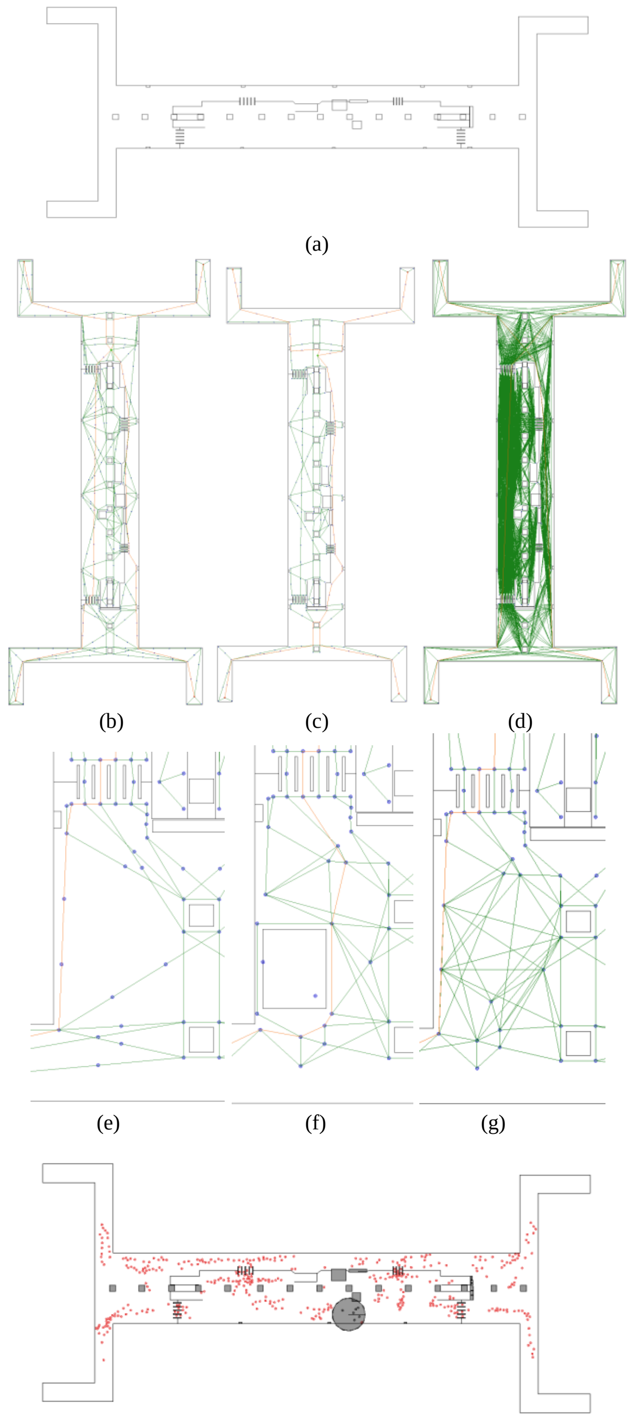


Figure. 5: (a) planar map of Jiedaokou Metro station. Black line represents obstacles and boundary of the map. (b), (c) and (d) are waypoint graph and path find result generated respectively by Balanced Waypoint Generation, Minimal Waypoint Generation and Brutal Waypoint Generation. Green lines represent edges of waypoint graph; the single green point is start position and 4 red points are 4 target positions; the red line is path found from start position to every end target position. (e), (f) and (g) are the waypoint graph and path-find result generate by Balanced method, respectively representing before new obstacle (the black hollow rectangle) insertion, after obstacle insertion and obstacle removal. The blue dots are waypoints, and the green lines are edges among waypoints. The red lines represent the path-find result. (h) Simulation of a dynamic environment with 400 characters and a sudden added obstacle (grey circle). After dynamic update the waypoint graph, characters find their new shortest ways to four exits at for corner of the map.

VII. Conclusion

This paper presented a novel waypoint generation and update strategy that facilitates navigation in planar environment. Unlike traditional way of test reachability via track line intersection, we considered the size of character and applied track area to test reachability. What's more, we implemented post process to sparsify graph's edge and set length limit and interpolate long edges into shorter ones, and this facilitates map refreshing in a high frequency when dynamic change happened. Experiments shows that the post process takes a little overhead while brings great efficiency into updating map, compared with brutal way. Finally, path-finding result shows that the path length on graph generated by balanced way is quite close to the theoretical shortest path and the path is more natural than minimal approach.

References

[1] Björnsson Y, Enzenberger M, Holte R, et al. Comparison of different grid abstractions for pathfinding on maps[C]//IJCAI. 2003: 1511-1512.

[2] Sturtevant N R. Benchmarks for grid-based pathfinding[J]. Computational Intelligence and AI in Games, IEEE Transactions on, 2012, 4(2): 144-148.

[3] Nash A, Koenig S. Any-Angle Path Planning[J]. AI Magazine, 2013, 34(4).

[4] AI game programming wisdom[M]. Cengage Learning, 2002.

[5] Tozour, P. (2003). Search space representations. *AI Game Programming Wisdom*, *2*(1), 85-102.

[6] Andrews, J. R. (1983). *Impedance control as a framework for implementing obstacle avoidance in a manipulator* (Doctoral dissertation, Massachusetts Institute of Technology, Department of Mechanical Engineering).

[7] Borenstein, J., & Koren, Y. (1989). Real-time obstacle avoidance for fast mobile robots. *Systems, Man and Cybernetics, IEEE Transactions on*, *19*(5), 1179-1187.

[8] Hagelbäck, J., & Johansson, S. J. (2008). The Rise of Potential Fields in Real Time Strategy Bots. *AIIDE*, *8*, 42-47.

[9] Nash A, Daniel K, Koenig S, et al. Theta*: Any-Angle Path Planning on Grids[C]//Proceedings of the National Conference on Artificial Intelligence. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007, 22(2): 1177.

[10] Ferguson D, Stentz A. Using interpolation to improve path planning: The Field D* algorithm[J]. Journal of Field Robotics, 2006, 23(2): 79-101.

[11] Welzl, E. (1991). *Smallest enclosing disks (balls and ellipsoids)* (pp. 359-370). Springer Berlin Heidelberg.

[12] Dijkstra, E.W.: A note on two problems in connexion with graphs.Numer. Math.1(1), 269–271 (1959)

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

[14] Wardhana N M, Johan H, Seah H S. Enhanced waypoint graph for surface and volumetric path planning in virtual worlds[J]. The Visual Computer, 2013, 29(10): 1051-1062.

[15] Wardhana N M, Johan H, Seah H S. Enhanced waypoint graph for path planning in virtual worlds[C]//Cyberworlds (CW), 2012 International Conference on. IEEE, 2012: 69-76.