A Star Algorithm on Quadtree and Its Application in Real-World Path Finding

Yonas Adiel Wiguna - 13516030
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indoneisa
ya@students.itb.ac.id

***Abstract*—One of path finding algorithm is A Star algorithm. However, the algorithm is difficult to be applied to real-world path finding. The possibilities of starting and destination node position is too many. Hence, we may solve this using A Star algorithm that run on Quadtree, so the number of nodes doesn't have to be large but still leave the freedom on chosing starting and destination node.**

Keywords—a star; quadtree; path finding; shortest path

I. Introduction

Path finding is one of interesting problems found in algorithm design. In daily life, we may need path finding for finding the shortest path from one location to other location. The size of the path is diverse greatly, ranging from within a building to between countries. The constraints usually the time it takes from one place to other place or simply the distance taken by following the path.

One of well-known algorithm is A* algorithm, which is working great in 2 dimensional map. Details about this algorithm will be dissccussed in later chapter. But this kind of algorithm still lack of some properties that occur in real-world situatuons. For example, a path may be farther than another path, but less in time. How we choose, how we optimize the complexity of the algorithm is an open question for everyone.

Other constraints is real-world map. For example, some old pixel RPG (Role Playing Game) use some kind of matrix to visualize its world. A player always located inside a cell in the matrix. The character won't be able to stand in boundary between two pixel, or other fragmented postion. Here, we can ensure the character always walk north, east, south, or west. We can go southeast, 30 degree clockwise from south, or other. Here, visualizing the path finding is super easy. Every cell inside the matrix will be treated as node or vertex in a graph. Adjacent cell will be reduced to edge in graph. Hence, any shortest path algorithm is an easy task to do.

However, in real-world, we may walk as we want, go into building that we want, walk 30 degree from clockwise from south, etc. We may not standing at the right position, or using real instead of integers. Here, the size of the proble increase rapidly – we cannot store the pixels as nodes.

This paper proposed new solution, an A* algorithm running on quadtree. Then, no need to worry about map shape because we may compress it into quadtree data structure. This data structure use divide and conquer concept on its tree. It used in two dimensional plane, so usually it is used for image processing. For example, lossy image compression to image encoding. Here, we use it to simplify real-world map into a graph that computer could understand and work properly.

II. Graph, Path Finding, and Quadtree

A. Graph

A Graph $G = (V, E)$ consists of V, a nonempty set of *vertices* (or *nodes*) and E, a set of *edges*. Nodes may be reduced from a location in map, a state, or other thing in graph theory. Each edge in E has two vertices associated with it, called its *endpoints*. An edge is said to *connect* its endpoints [1]. Two vertices u and v are *adjacent* if and only if there is at least one edge $e \in E$ that have u and v as its endpoints, and they are *connected* if there is some path from u to v, or path from v to u. Then, e can be called *incident* with vertices u and v and representated as (u, v).

By definition, there are some special graphs. A graph where all of its vertices are adjacent to all other vertices are called *complete graph*. *Tree* is a special graph where every pair of vertices are connected, and $|E| = |V| - 1$. There are some other special graphs more, but won't be mentioned in this paper.

Then, graph can be categorized by its properties. First, for every edges, we may have a value assign to it. Then, the graph will be *weighted graph*. On the other hand, *unweighted graph* don't have any value assign o its edges. By some observastion, we may tell that unweighted graph have some behavior as weighted graph with same weight among all edge. In this paper, we are interested in weighted graph. Then, we may convert unweighted graph to weighted graph by pick any positive value and assign it to every edge.

Then, we may have a *directed graph* and *undirected graph*. As the name implies, directed graph has direction in its edges. Edge (u, v) means there is edge from u to v, but it doesn't ensures that there is an edge from v to u. Node u will be called *source* and node v will be called *destination*. So undirected graph means edge (u, v) implies there is exist both path from u to v and v to u.

One can present graph with some representation. There are some representation: adjacency list, adjacency matrix, edge list,

and some more representation. Other representations are not mentioned, because they aren't relevant to this paper.

Adjacency list is list of nodes, where every node points to list of nodes that are adjacent to the node. Then, a vertex u is connected to vertex v if v is exist in u's adjacency list. In undirected graph, the edge will apear to times, in u and v's list. If the graph is weighted graph, then we noted store not only v in u's adjacency list, but also the weight itself. So the adjacency list will have tuple of (v, w) where v is the adjacent vertex and w is the weight of edge (u, v). In programming, this representation is best used to traverse a path from source to destination, such as BFS, DFS, or Dijkstra algorithm. This paper mostly will use this algorithm.

Adjacency matrix is a matrix with dimension of number of vertices. For example, if A is adjacency matrix of graph G, then $A_{u,v}$ will denote the existence of edge (u, v). It means, $A_{u,v}$ will have value 0 if there is no edge (u, v) but 1 if there is edge (u, v) in E. The matrix will be binary matrix. If the graph is undirected graph, $A_{u,v}$ will equal to $A_{v,u}$. If the graph is weighted, we store the weight as value of $A_{u,v}$. This means if there is no edge exist from u directly to v, then $A_{u,v} = \infty$. In programming, this representation is best used to storing near-complete graph, which is if we pick randomly u and v, there is a high chance we have edge (u, v) in our graph.

Edge list is a list that contains all edges as tuple (u, v) which is denotes the existence of edge (u, v) in the graph. If tuple (u, v) doesn't exist in the edge list, u is not adjacent to v. If the graph is directed, we will differentiate the meaning of (u, v) and (v, u). If the graph is weighted, we will use (u, v, w) instead of (u, v) where w is the cost from u to v. In programming, this representation best used by Kruskal algorithm where one want to compute MST (Minimum Spanning Tree) of given graph. Minimum spanning tree is a tree made from undirected weighted graph by removing its edge so the sum of leftover edges is the mnimum possible one can get.

A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ if and only if V' is subset of V, and E' is subset of E [1]. A subgraph can be generated with removing zero or more of its edges and removing zero or more vertex with its incedental edges.

A path is a squence of vertices with every two adjacent vertices in this sequence have valid edge incedent with both of them. A path is said simple path if the sequence can be traversed without using any edge more than once. A circuit is a path where the starting vertex is the same as goal vertex. Again, a simple circut is a circuit where the sequence can be traversed without using any edge more than once.

B. Path Finding

Path finding is a process done in order to find a path from starting vertex to goal vertex. Usually, the path needed is the shortest possible. Given a graph G, s as source, and t as destination, we need to find whether or not exist a path from s to t. If the answer is exist, then we need to output the path.

One of well-known graph traversal algorithm is Breadth First Search and Depth First Search. The algorithm is as follows:

1) Mark starting node as current node,
2) For every node adjacent to current node, if it predecessor is not set.
   a) Mark current node as its predecessor
   b) Mark it as current node and process from step number two.

The pseudocode is as follows:

```
def traverse(graph, start, dest):
   dist = new Int(graph.node.length) = {INF}
   dist[start] = 0
   list = new List()
   list.push({ start })

   while (! list.empty()
            && dist[dest] == INF):
      cur = list.pickOne()
      for (next in graph.node[cur].adj):
         if (dist[next] == INF):
            dist[next] =
               dist[cur] + next.dist
            list.push(next.node)

   return dist[dest]
```

However, there is a decision we should make on above algorithm. If there is processing queue list for every pending processed node, should we process the last added or the first added to the list? This will differentiate BFS from DFS: BFS use the first added and DFS use the last added to the list. The implementation will use queue and stack and will be left to reader.

Backtracking the answer will be easy, which is looping the predecessor of destination node until we found source node. However, the optimal solution is not guaranteed. If the graph is unweighted, then DFS won't give the optimal solution, meanwhile BFS will return the optimal solution. The proof is left to the reader. However, if the graph is weighted, than BFS algorithm won't return the optimal solution, because we can't pick the right node to be processed next.

It means new algorithm is required. Dijkstra algorithm is the best solution for this problem. The algorithm is as follows:

1) *Initiate empty list*
2) *Initiate distance of all node as infinite.*
3) *Add tuple of starting node and 0 as new distance to list.*
4) *Get dan remove node from list with minimum.new distance.*
5) *If the list is empty, exit program.*
6) *If distance of node is not infinite, repeat to step 4.*
7) *Set the distance of node to .new distance.*
8) *For all nodes adjacent to current node, add tuple of next node and their new distance from current node to the list.*

However, the list must be able to sort every insertion and removed quickly enough. Hence, we can use priority queue as list object. The pseudocode is as follows:

```
def dijkstra(graph, start, dest):
   dist = new Int(graph.node.length) = {INF}
   pq = new PriorityQueue()
   pq.push({ start, 0 })
   /* priority queue will always sorted
      ascending by second value */

   while (! pq.empty()
            && dist[dest] == INF):
      cur = pq.pop()
      if (dist[cur.node] == INF):
         dist[cur.node] = cur.dist
         for (next in graph.node[cur.node].adj):
            pq.push({
               next.node,
               dist[cur.node] + next.dist})
   return dist[dest]
```

This algorithm is good enough for most of cases, but it is pretty bad in real-world situations. For example, if the destination node will have distance x, algorithm will traverse to all every node with distance less than x, even though we know the shortest path won't use this path. For example, take this illustration:

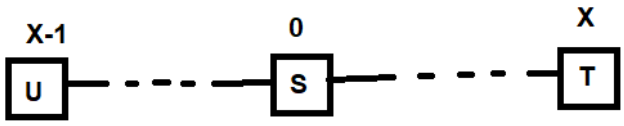


Figure 1 Example of bad case for djkstra algorithm

If this a dijkstra on real-world map, and s is start node, t is the destination, this example is one of bad case for dijkstra. From starting node s, we will traverse through many node until get to u, traverse to u, and then we will be able to process t. However, because this is exampleof real-world map, we know by intuition we won't have to visit u if we want to go to t. We don't have to go to west if we want to go to east. Of course, this intiuition usually only apply if x is big enough.

Then we will observe this intuition to improve the algorithm. The advantages of real-world map is we base the map from real coordinate. Then we can come to this lemma.

Lemma 2.1. The minimum distance required of two nodes in real-world map is length of straight line that connect those two nodes.

This lemma is very useful to our next algorithm. Dijkstra will pick the node which is shortest from source. Here, we can pick the node which is shortest from source and (maybe) will have shortest path to destination. The word maybe is used because we don't know for sure whether it will be shortest or not, but we know for sure the minimum distance it will take to destination node using Lemma 2.1.

Now we modify the dijkstra to consider not only the distance from source, but also the minimum distance to destination:

```
def sqr_euclid_dist(graph, i, j):
   return
      (graph.node[i].x - graph.node[j].x)^2 +
      (graph.node[i].y - graph.node[j].y)^2

def astar(graph, start, dest):
   dist = new Int(graph.node.length) = {INF}
   pq = new PriorityQueue()
   pq.push({ start, 0 })
   /* priority queue will always sorted
      ascending by second value */

   while (! pq.empty()
            && dist[dest] == INF):
      cur = pq.pop()
      if (dist[cur.node] == INF):
         dist[cur.node] = cur.dist
         for (next in graph.node[cur.node].adj):
            pq.push({
               next.node,
               dist[cur.node] + next.dist +
               sqr_euclid_dist(
                  graph, cur.node, next.node)})
   return dist[dest]
```

We name it astar, as A* algorithm behavior that is the same as above algorithm. Then, we may compute the real-world path finding much better than dijkstra algorithm.

C. Quadtree

Quadtree is a data structure used in two dimensional data. Quadtree is widely known for lossy image compression, which means we may reduce size of an image file, but loss in quality too. A quadtree data structure is recursive, with single quadtree may refer to zero or 4 other quadtree. For ease, we done these 4 subquadtree as quadrant. For example, if we have black and white image as follows:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
0	0	1	1	1	1	0	0
0	1	1	1	1	1	0	0
0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

Figure 2 Example of black and white image (1 stands for black, 0 stands for white)

Notice that the upper left is full of zero. The idea of quadtree is note all those pixel by zero once, not for every pixel of them. But if the quadrant doesn't have the same color, we divide it again to smaller quadrants. We may repeat this untill the smallest unit in our image. However, one pixel in an image should have full black or full white, so the rule still holds.

<table>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><td colspan="4" rowspan="2">0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td colspan="2" rowspan="2">1</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td rowspan="2">0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Figure 3 Example of quadtree representation from Figure 2

For lossy image compression, when we make a subquadtree, we calculate the mean of its color. Then if the variance isn't too big, we assume they are all have same color, which is their median.

Now we know how quadtree is useful for encoding and image, we may use it for encode our map. Again, we gain benefit of using real-world problem. Usually, a building has polygon top-view-shape. To be precise, most of them are rectangle. Then, we may encode the image of map into image of rectangles.

Dividing the image until finding precise outline will be exhausting, then we only need to define the *resolution* of quadtree. Then the minimum square radius in quadtree will be this resolution. Then, we will rounding four edges of the rectangles that will be drawn to outer or inner, depends on its location.

For example, author will use Google Satllite image of Bandung Institute of Technology Library and Center of Advance Sciences Building:

(a)

(b)

(c)

Figure 4 (a) original image from google image, (b) quadtree boundary of image, (c) building corresponding in quadtree. Note: for clarity of the example, many deatils are ignored.

We encode and decode the shape using divide and conquer strategy. The algorithm for general quadtree is left to reader because the quadtree that will be used in this paper has slightly difference with the general quadtree encoding algorithm.

To code the quadtree that will be used in this paper, we need to define only two procedures: `fillRect(left, top, right, bottom, color)` and `isFullColor(subQuadtree, color)` which will return true if in the given subQuadtree, all color is the given color. To di this optimally, we must keep our Quadtree data structure as simple as possible. That means, if a quadtree divided to 4 quadtree that all have same color, immediatly delete them thus the height of current quadtree is reduced (be a leave of the tree).

The algorithm is to fill rectangle inside a quadtree or subquadtree as follows:

1) If subquadtree bound is within rectangle bound, color this subquadtree and delete its child then return.

2) If rectangle bound is out of subquadtree bound, don't color anything and return.

3) If current subquadtree doesn't have any child, create 4 child with same color as current subquadtree.

4) Find the middle horizontal and vertical line.

5) Fill every quadran with same color.

6) If in the end all quadrans are leaves and have the same color, make current subquadtree as leaf with quadrans' color.

Above algorithm will ensure the height of any quadtree will be the minimum it can get. With this algorithm, we may color a rectangle or *decolor* a rectangle. The complexity of quadtree can be improved, but will not be discussed in this paper for simplicity [2]. Please refer to references.

III. A Star Algorithm on Quadtree

The rest of problem is make A* algorithm run in quadtree. The idea is simple, that is we only need to make every blank quadtree as node. The node position is its x and y position. To do that, we need to build all the nodes. To maintain the complexity, we need to do this in divide and conquer. This can be easily done with recursively calls `buildNode(nodes, left, top, right, bottom, subQuadtree)`. The algorithm is as follows:

1) If current subQuadtree is leaf, push the middle x and y to nodes, together with its color, then return

2) .Otherwise, call buildNode for top-left, top-right, bottom-left, bottom-right.

The algorithm for building the node is easy. However, we need more attention on building the adjacency list. Again, to maintain the complexity, we need to do this by divide and conquer. However, this divide and conquer is different: it heavy on the combine step.

Assume that we have build the adjacency list in all four quadrants of a subquadtree. Then we need to combine all those adjacency list and combine it with adjacency list of 4 neighboring quadrants. It means, match all right-most node in top-left quadrant to left-most top-right quadrant, match all bottom-most top-right quadrant to top-most bottom-right quadrant, left-most bottom-right quadrant to right-most bottom-left quadrant, and top-most bottom-left quadrant to bottom-most top-left quadrant.

We will call `buildEdge` until we reach both leaves. However, we need to keep in mind that subquadtree size may have different size. We need to stop one of the quadrants if it is leaf already. The illustration as follows:

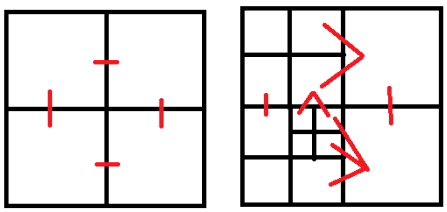


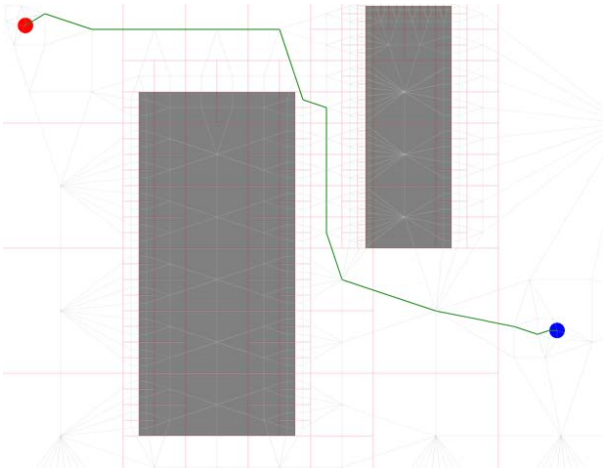
Figure 5 Combining step on building adjacency list of quadtree. The red line indicates the area that needs to be considered. The right image show how combining different size quadtree.

With little observation, we may notice that actually we need to do only 2 things: combining top and bottom, combining left and right. If we call it with right arguments, we will get the right results. So we create `buildEdgeTopBottom(topBoundary, bottomBoundary, subQuadtreeTop, subQuadtreeBottom)` and `buildEdgeLeftRight(leftBoundary, rightBoundary, subQuadtreeLeft, subQuadtreeRight)`.

For both of building edges left-right and top-bottom, we always recursively calls it children if both of subQuadtree have children. If both of them are leaf, we combine it right away. If one of them is leaf, then keep the leaf and keep dividing the subQuadtree which has children.

Details of algorithm implementation are not given. The implementation can be checked at author's implementation in link in appendix or as an exercise to reader.

The rest of the program is user interface implementation. The implementation won't be deatiled in this paper. The program can be tried interactively in link in appendix of this paper. Some of the screenshot will be shown here. Red dots indicates starting node and blue dots indicates destination node. The blurry red lines are quadtree boundary, meanwile blurry grey lines are edges in adjacency lists.



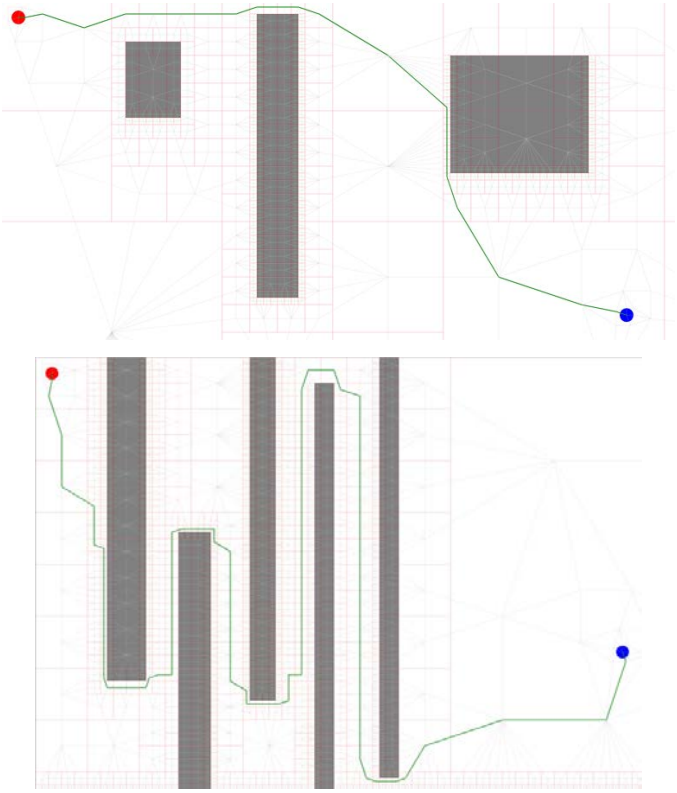


Figure 6 Example of program execution

IV. Appendix

The author's implementation of algorithm discussed in this paper can be found in author's github repository (https://github.com/yonasadiel/a-star-on-quadtree). It uses JavaScript for computing and visualiing the answer. All algorithm is the same as given in this paper, but may be dfferent on implementation.

Live interactive demo can be accessed from the readme file or directly to https://yonasaiel.github.io/a-star-on-quadtree. The implementation uses jQuery 3.2.1 library for interaction with HTML5 elements.

The image used in this paper is taken from Google Satellite (https://www.google.co.id/maps/@-6.8881637,107.6110852,218m/data=!3m1!1e3).

V. Acknowledgment

The author wants to thank Dr. Ir. Rinaldi Munir, MT as the lecturer of Algorithm Design IF2211 course in author's class. The author also would thank Bandung Institute of Technology for its providence of access to IEEE documents and papers. Many inspirations and references affect author works.

References

[1] K.H. Rosen, Discrete Mathematics and Its Application, 7th ed. New York: McGraw-Hill, 2012, pp. 641-802.

[2] B. G. Mobasseri, "A new quadtree complexity theorem," Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems, The Hague, 1992, pp. 389-392.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 14 Mei 2018

Ttd

Yonas Adiel Wiguna

13516030