**Dijkstra’s Shortest Path Algorithm**

The Dijkstra’s shortest path algorithm is the most commonly used to solve the single source shortest path problem today. For a graph G(V, E), where V is the set of vertices and E is the set of edges, the running time for finding a path between two vertices varies when different data structure are used. This project uses binary heap to implement Dijkstra’s algorithm although there are some data structures that may slightly improve the time complexity, such as Fibonacci heap that can purchase time complexity of O(V\*log(V)) [6].

Dijkstra’s shortest path algorithm for each u G:

d[u] = infinity; parent[u] = NIL;

End for

d[s] = 0; // s is the start point H = {s}; // the heap

while NotEmpty(H) and targetNotFound: u = Extract\_Min(H);

label u as examined;

for each v adjacent to u:

if d[v] > d[u] + w[u, v]:

d[v] = d[u] + w[u, v]; parent[v] = u; DecreaseKey[v, H];

Time Complexity:

The run time of first for loop is O(V). In each iteration of the while loop, Extract\_Min of the heap is logV. The inner for loop iterates each adjacent node of the current node, the total run time is O(E). Therefore, the time complexity of this algorithm is O((V + E)\*log(V) = O(E\* log(V)). The correctness of this algorithm is well proved in [6].

As the number of nodes in a graph increases, the running time of the applied algorithm will become longer and longer. Usually, a road network of a city has more than 10^4 nodes. A fast shortest path algorithm becomes more desirable.

Basically, the structure of road networks is relative simple. They are large scaled, sparse and connected graph. When the Dijkstra algorithm is used to find the shortest path, it starts search from the start point and spreads as a circle until the radius arrives the destination. Most searches at the area opposite the direction of destination are useless. M. Fu et al. [4] described an optimal approach to find shortest path for Vehicle Navigation System by physically cutting off area within which the shortest path is not supposed to appear.

|  |  |  |
| --- | --- | --- |
|  | T2 |  |
| Rec1 D  T1  S Rec2 | | |

Search Area of Dijkstra and Restricted Search Algorithm (S: Start point, D: Destination)

Instead of search the entire circle, the Restricted Search Method only search with the small area of the remaining part of rectangle Rec2 cutting off by the two bold straight lines. The rectangle Rec1 has the straight line of S and D as a diagonal and Rec2 is a rectangle extended from Rec1 by a threshold T2. The two straight bold lines parallel the

straight line of SD with a distance of T1. T1 and T2 are two variables that need to be decided to ensure that the optimal path is included within the restricted area. They usually range from 500m to 1500 m.

This project uses the following algorithm to achieve this goal. Restricted Search Algorithm:

for each u G:

d[u] = infinity; parent[u] = NIL;

End for

d[s] = 0;

H = {s};

while NotEmpty(H) and targetNotFound: u = Extract\_Nin(H);

label u as examined;

for each v adjacent to u:

if outOfRange(v), then continue; if d[v] > d[u] + w[u, v], then

d[v] = d[u] + w[u, v]; parent[v] = u; DecreaseKey[v, H];

Procedure outOfRange(Constraint Area A, Vertex v):

//A is a polygon given;

//v is a Vertex being checked;

Make a straight-line L from v to the right of v; Counter = 0;

For each edge e of A

if L intersects with e

increase Counter by one; if Counter is even

return true;

else

return false

Time Complexity:

Suppose the nodes in the road network are distributed evenly.

Let density be C,

V\*: the number of vertices examined by using Dijkstra algorithm. V: the number of vertices examined by using this algorithm.

Soptimal: searched area by using this algorithm. Sdij: searched area by using Dijkstra algorithm. K1, K2: the factors of threshold.

Then, the ratio P can be used to describe the improvement of efficiency, P = V/V\*

= C\* Soptimal / C\* Sdij

<= 2 \* T1 (R + T2 \* √2) / πR²

= 2 \* K1\*R(R + K2\* R\*√2) / πR²

=2 \* K1(1 + √2 \* K2) / π

Since K1 and K2 are small number, the great improvement can be achieved. For instance, if K2 = 0.4, P ≈ K1.

**Correctness:**

There are two problems exist in this approach.

1. The fixed threshold may not get the solution properly. As the distance from start point to the destination increases, the shortest path more likely spreads wider from the straight line of SD.
2. The city traffic lines have different categories with different driving speed. The high way may be beyond the restricted area but in the shortest path.

To achieve a better solution, this project uses relative thresholds instead of the fixed ones, called factor K1, K2 (threshold / length of SD). To simplify the problem, they have same value in the implementation. The threshold proportionally increases with the distance from start point to the destination in a selected factor. The second problem could be solved by using logical position instead of physical location for each node. For example, upon a node being examined, its logical position will be the accumulation from the logical location of its parent by the cost of the edge between them. The cost of an edge is logical distance that relative to physical distance and road category The

logical location of start point is the same as its physical location. All nodes being examined use their logical position to decide if they are within the restricted area. This ensures that the nodes on different roads can be treated equally in searching. However, due to the time constraint, this project will not implement it in the program.

This algorithm actually uses the Dijkstra within the restricted area. It is obvious that a shortest path in this area will be found if the path exists. However, this shortest path may not be the shortest one in the whole area. Thus, it is optimal path with restricted area.

**A\* Search**

The A\* algorithm integrates a heuristic into a search procedure. Instead of choosing the next node with the least cost (as measured from the start node), the choice of node is based on the cost from the start node plus an estimate of proximity to the destination (a heuristic estimate). F. Engineer [2] described this approach to solve the problem of optimal path finding.

This project uses Euclidean distance as estimated distance to the destination. In the searching, the cost of a node V could be calculated as:

f(V) = distance from *S* to V + estimate of the distance to *D.*

= d(V) + h(V,D)

= d(V) + sqrt( (x(V) – x(D))² + (y(V) – y(D))²)

where x(V), y(D) and x(V), y(D) are the coordinates for node V and the destination node D.

The A\* Search algorithm: for each u G:

d[u] = infinity; parent[u] = NIL;

End for

d[s] = 0;

f(V) = 0;

H = {s};

while NotEmpty(H) and targetNotFound: u = Extract\_Min(H);

label u as examined;

for each v adjacent to u:

if d[v] > d[u] + w[u, v] , then

d[v] = d[u] + w[u, v]; p[v] = u;

f(v) = d[v] + h(v, D); DecreaseKey[v, H];

Time Complexity:

This algorithm does not improve worst case time complexity, but it improves average time complexity. The shortest path search starts from start point and expands node that goes towards the destination. Therefore, the run time is much shorter than the Dijkstra’s algorithm.

Correctness:

The algorithm uses the same approach as Dijkstra’s except that it uses accumulated cost of edge plus the Euclidean distance from current node to the destination. This value is used to decide the position of a node in the min heap. The one with smallest value will be selected and removed from the heap. In the implementation, this value only affects the searching order. It doesn’t modify the edge weights and accumulated distance. The accumulated distance is updated as the same way as Dijkstra when a node relax.

Therefore, this algorithm is same as Dijkstra and it is correct.