## Map Routing

**Dijkstra's shortest path algorithm**

Thinh Nguyen  
 Computer Engineering  
 San Jose State University  
 oohungthinh97oo@gmail.com

Chau Doan  
 Computer Engineering  
 San Jose State University  
chau.doan@sjsu.edu

Phuong Ngo  
 Computer Engineering  
 San Jose State University  
phuong.ngo@sjsu.edu

ABSTRACT

Dijkstra's algorithm is a classic solution to the shortest path problem. It is described in Sedgewick, Chapter 21. The basic idea is not difficult to understand. We maintain, for every vertex in the graph, the length of the shortest known path from the source to that vertex, and we maintain these lengths in a priority queue. Initially, we put all the vertices on the queue with an artificially high priority and then assign priority 0.0 to the source. The algorithm proceeds by taking the lowest-priority vertex off the PQ, then checking all the vertices that can be reached from that vertex by one edge to see whether that edge gives a shorter path to the vertex from the source than the shortest previously-known path. If so, it lowers the priority to reflect this new information.

1. Introduction

We will implement the classic Dijkstra's shortest path algorithm and optimize it for maps. Such algorithms are widely used in geographic information systems (GIS) including MapQuest and GPS-based car navigation systems (Sedgewick, 2004).

We will optimize Dijkstra’s shortest path algorithm so it can process thousands of queries in sub-linear time. Our goal is to solve this problem without using excessive memory space. There are many ways to improve Dijkstra’s shortest path algorithm. The first approach is to stop the search when we find the shortest path for a query. For the next queries, we can save time by only reinitializing the vertices’ distance values that have been changed in precious queries.

We can also improve it further by taking to account the coordination of the sink when calculating the priority of each vertex. What we want is for the algorithm to priority the vertices that is both close the source and to the sink. By applying the triangle properties of a Euclidean network. The distance from s to do is never greater than the distance from s to x plus distance from x to do. So instead of relaxing an edge v-w by updating wt[w] to wt[v] plus the distance from w to v, we “update wt[w] to be the sum of wt[v] plus the distance from v to w *plus* the Euclidean distance from w to d *minus* the Euclidean distance from v to d” (Sedgewick, 2004). This would improve the performance but preserve the correctness.

1. Design methodology
2. Formula

* Distance between 2 nodes will be calculated by the following formula:

Distance(node1,node2):

* First improvement is every time we pop a vertex out of the queue, we know the its shortest path to the source will not change anymore, and so it is the target vertex, then we can stop the algorithm. When we initialize the vertices, we only have to do that for vertices that we visited, and we will have list to keep track of the visited vertices.
* Second improvement is that instead of relaxing an edge v-w by updating wt[w] to wt[v] plus the distance from w to v, we “update wt[w] to be the sum of wt[v] plus the distance from v to w *plus* the Euclidean distance from w to d *minus* the Euclidean distance from v to d

new\_dist = current.get\_distance() + current.get\_weight(next) + dist(next, target) - dist(current, target)

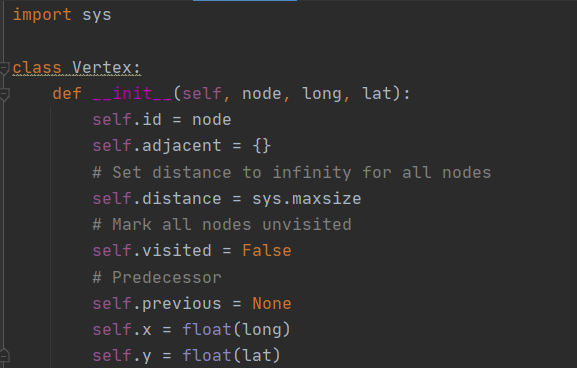
1.1 Class Vertex

Figure 1.1.a. Constructor of class Vertex

* In order to implement Dijkstra's algorithm, we have to create class called Vertex. In this class, we have a constructor with 3 different arguments which are node, long and lat. All distances will be from node to other nodes wil be set to infinity, and all nodes will be marked as unvisited.



Figure 1.1.b. Overload greater than in class Vertex

* Figure 1.1.b shows the function that needed to be overload in class Vertex for heapq



Figure 1.1.c. add\_neighbor function in class Vertex

* Figure 1.1.c shows function add\_neighbor which set the weight between 2 nodes.



Figure 1.1.d. get\_id function in class Vertex

* Figure 1.1.d shows function get\_id which returns id of a node.

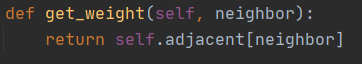


Figure 1.1.e. get\_weight function in class Vertex

* Figure 1.1.e shows function get\_weight which returns weight between 2 nodes.



Figure 1.1.f. set\_distance function in class Vertex

* Figure 1.1.f shows function set\_distance which to set distance



Figure 1.1.g. get\_distance function in class Vertex

* Figure 1.1.g shows function get\_distance which returns the distance



Figure 1.1.h. set\_previous function in class Vertex

* Figure 1.1.h shows function set\_previous which set argument prev to self.previous



Figure 1.1.i. set\_visited function in class Vertex

* Figure 1.1.i shows function set\_visited which sets nodes to visited (Boolean TRUE)



Figure 1.1.g. overload str() function in class Vertex

* Figure 1.1.g shows function \_\_str\_\_ which controls how the object gets printed.
  1. Class graph

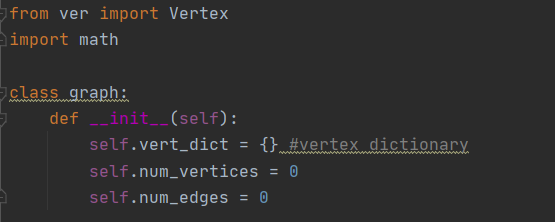


Figure 1.2.a. constructor of class graph

* Create a class graph in order to store all vertices with a default constructor which stores all the vertices into a dictionary and count all the number of vertices and all edges.

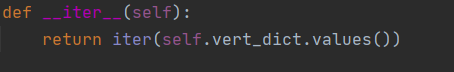


Figure 1.2.b. iter function of class graph

* Iter function returns an iterator for the given object.

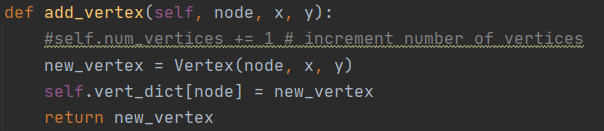


Figure 1.2.c. add\_vertex function of class graph

* Figure 1.2.c shows add\_vertex function which adds a new vertex into dictionary.

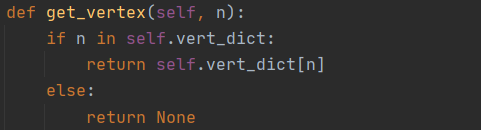


Figure 1.2.d. get\_vertex function of class graph

* Figure 1.2.d shows get\_vertex function which returns a vertex if it exists in the dictionary

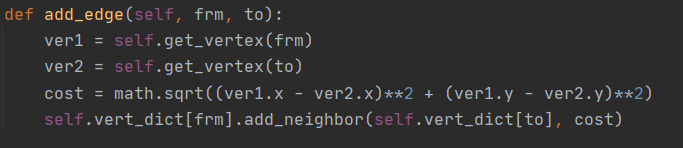


Figure 1.2.e. add\_edge function of class graph

* Figure 1.2.e shows add\_edge function which calculate the edge between 2 nodes.



Figure 1.2.f. get\_vertices function of class graph

* Figure 1.2.f shows get\_vertices function which returns vertices

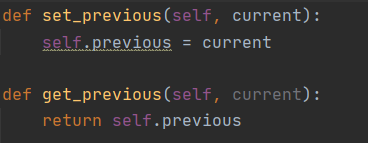
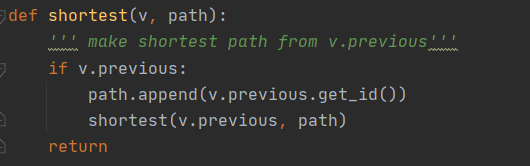
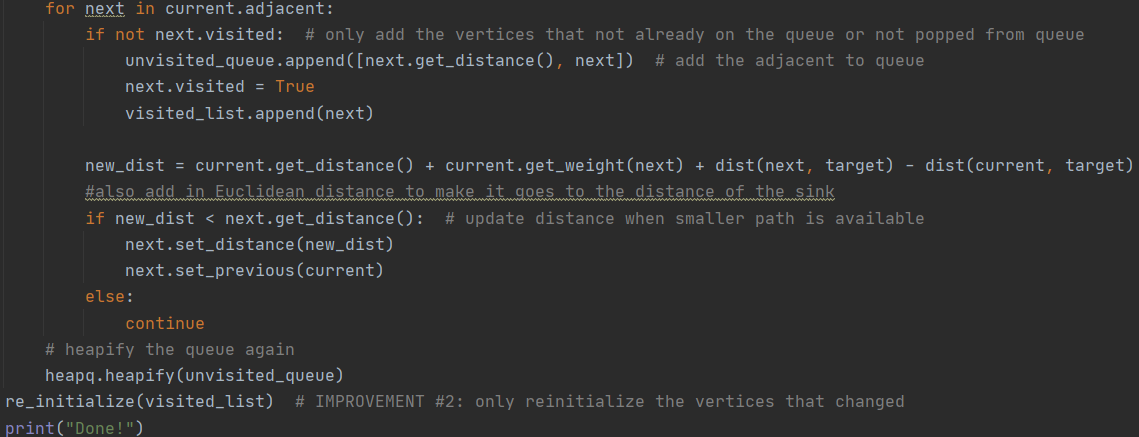


Figure 1.2.g. set\_previous and get\_previous function of class graph

* Figure 1.2.g shows set\_previous and get\_previous functions which set the current node to previous node and return previous node.

1.3. Implementation of Dijkstra’s shortest path algorithm.



Figure 1.3. a. Function to make shortest path from v.previous

* Figure 1.3.a shows shortest function which gathers predecessor starting from the target node.

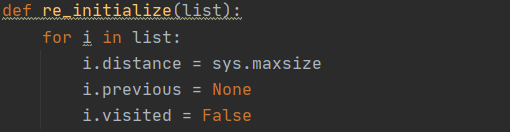


Figure 1.3.b. re\_initialize function

* Figure 1.3.b shows re\_initialize function which only reinitializes the vertices that changed



Figure 1.3.c. dist function

* Figure 1.3.c shows re\_dist function which takes 2 arguments (2 nodes) and calculates the distance of 2 nodes in 2D.

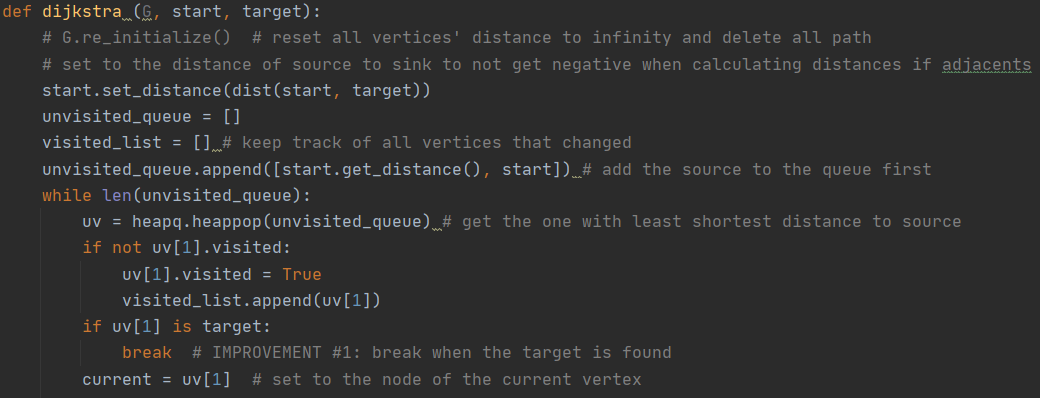


Figure 1.3.d. first half of dijkstra function

Figure 1.3.e. second half of dijkstra function

1. Testing.

In this part, we created several test case to test our program to make sure it functional correctly. In this test, use a simple map to test does all the edge, vertex, are store correctly and does the Dijkstra’s detect the correct path.

We used the Unittest library for this section.

For the first test, we will add vertex to the graph and test all function in the graph class.

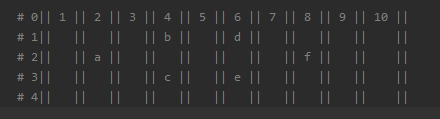


Figure 2.1 Example vertex coordinate for this test.

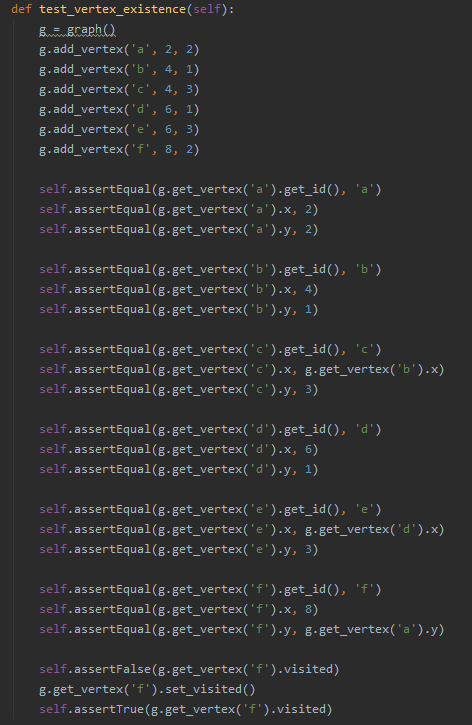


Figure 2.2 Testing the existence of the vertex.

* First, we tried to add to the vertex we have from Figure 2.1 corresponding with it coordinate by add\_vertex(vertex name, x coordinate, y coordinate) function.
* Second, we test each vertex to check whether the vertex exist by get\_vertex(vertex name) function. If it exists, it will return the vertex. If not, it returns None.
* Then, we test it name, x coordinate and y coordinate to check whether the function return the correct value.
* Finally, we test the visit status of the vertex.



Figure 2.3 Testing edge and distance between vertexes.

* From the example of test 1, we connect the vertex by add\_edge() function.
* Then, we test the distance by using get\_weight(target) function and compare with the actual distance. (The distance will be calculate by using Pythagoras theorem)
* Finally, we test all the edge corresponding with the vertex by print the information of the vertex and catch the output to compare with the actual edge.

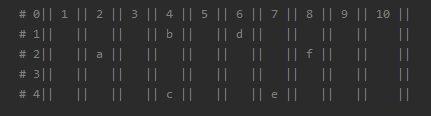


Figure 2.4 Modify example to get various distance.

We modify the example which use for the last test to have different distance so that we test the shortest path more accurate.



Figure 2.5 Testing Dijkstra shortest path.

* First we add all the vertex with the example in the Figure 2.4.
* Then we add all the edge to the graph.
* Finally, we call the Dijkstra function to get the shortest path and compare it with the actual shortest path (The function will be call two time to make sure all the status will be reset at the begin of the function an return the correct answer)

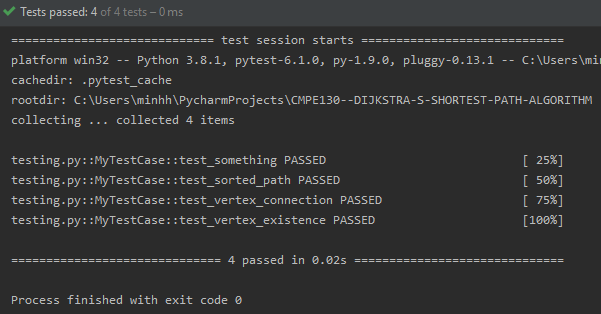


Figure 2.6 Testing result.

* All the test passed. We can conclude that our program function correctly.

*Performance Test:*

For the performance test, we run the original Dijkstra’s algorithm, Dijkstra’s with the first improvement, and Dijkstra’s with both improvement a hundred time each and then calculate the average to understand how much faster the improved algorithm is. Each time, the vertices for the source and sink are generated randomly. The test file contains 87575 vertices and 121961 edges, and not every vertex is connected to each other. The results indicate that by stopping the algorithm when encountering the target vertex, we were able to make it faster by 20% (Figure 2.7). The final improve algorithm, is even faster, it cuts down the processing time by about 48% (Figure 2.8). That is promising results. The only thing we were not able to test were how the data size would affect the processing time, because there were only on test file provided. So more testing is needed to understand average time complexity of the improve algorithm.

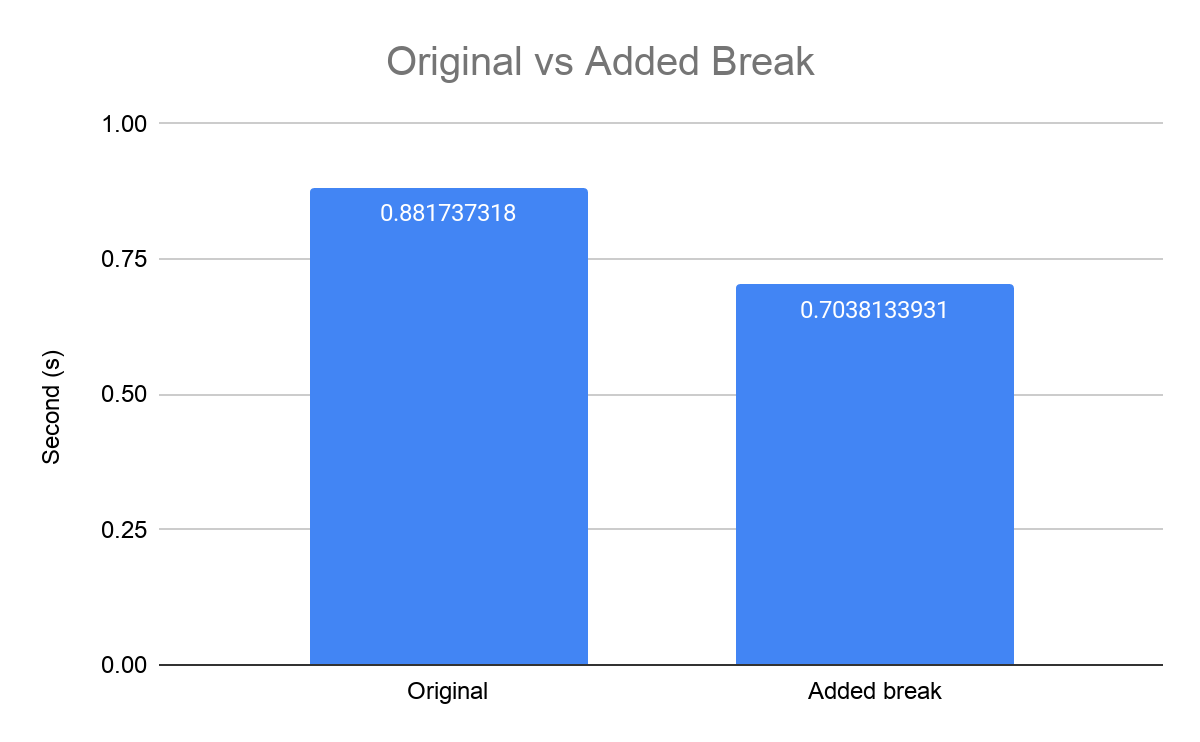


Figure 2.7 Processing time of original algorithm vs added break algorithm for one query

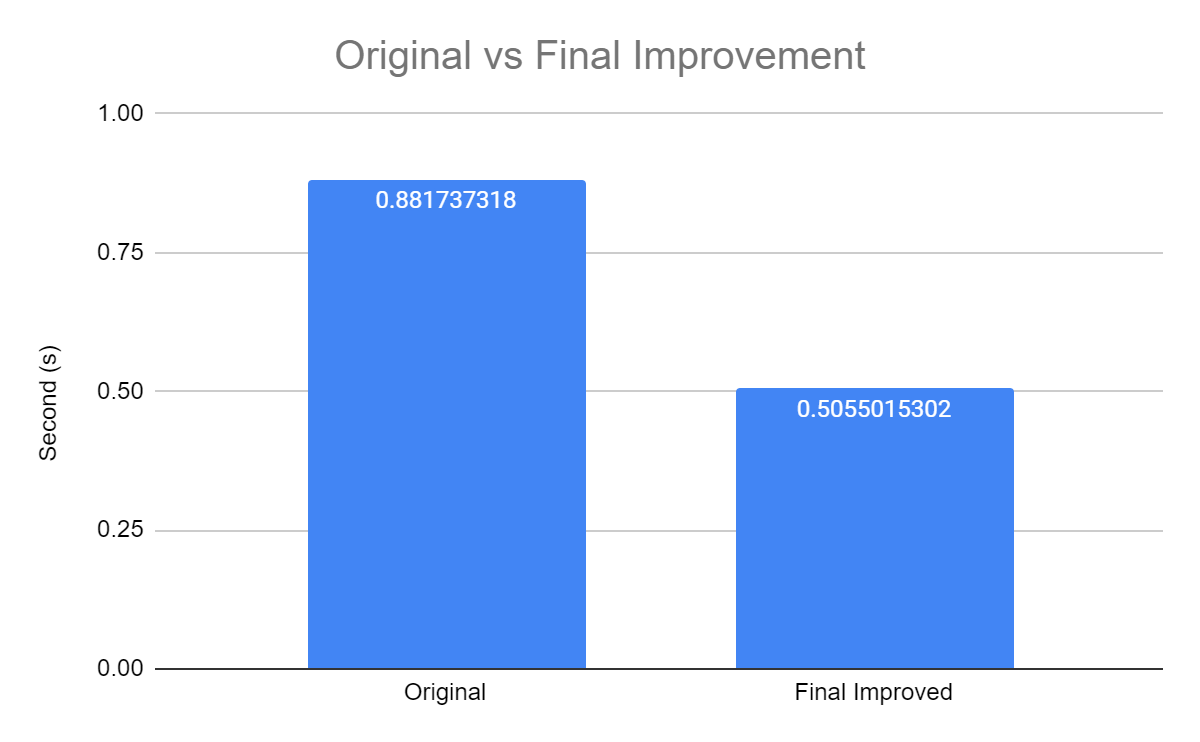


Figure 2.8 Processing time of original algorithm vs final algorithm for one query

1. ACKNOWLEDGMENTS

We want to thank Mr. Sedgewick and Mr. Wayne because we were able to complete this project by using the hints provided on the problem website.

REFERENCES

[1] Sedgewick, B., & Wayne, K. (2004). COS 226 Programming Assignment: Map Routing. Retrieved November 05, 2020, from https://www.cs.princeton.edu/courses/archive/spring04/cos226/assignments/map.htmlConference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00