Project SSP

Nick King

The problem which we are attempting to solve is that of the second shortest path through a grid. There are many different methods of solving this problem, but we are specifically asked to find the shortest path and the second shortest path between a source and destination node. At first sight, running a modified heap based Dijkstra’s algorithm multiple times to find the two shortest paths seems the most appealing as it fits the description of the problem more accurately than a dynamic programming approach. However, we will examine the dynamic programming approach due to the grid nature of our problem, creating an algorithm that finds the second path, and determine via asymptotic time complexity which algorithm better suits our needs. We are expected to deal with grids having dimensions bound asymptotically by 10000, so we must be sure our algorithm will run efficiently under these input conditions.

**Adjusted DSPA Algorithm**

When adjusting Dijkstra’s algorithm to find the second shortest path using a heap structure, we must implement a priority queue which holds <V,dist[V]> pairs, or the vertex and the shortest distance to that vertex. We must also keep track of another set of variables that contain the information for the second path, allocating space for another array.

int Dijkstra(grid, source)

set<int,int> heap

for each vertex in the graph

dist[i] 🡨 INF //initialize distance values to infinity

dist2[i] 🡨 INF

prev[i],prev2[i] 🡨 INF

heap.insert(i,dist[i]) //insert vertex-distance pair into the heap

dist[source],dist2[source] 🡨 0

while heap is not empty

u 🡨 heap.extract\_min() //takes highest priority item out of heap and assigns it to u

for all right or down neighbors (v) of current vertex (u)

alt 🡨 dist[u] + grid[u][v]

alt2 🡨 dist[u] + grid[u][v]

if alt < dist[v] //if distance from curr is less than node’s distance from source

dist2[v] = dist1[v] //stores second smallest value

prev2[v] 🡨 prev[v] //store prev value for ssp array

dist[v] = alt

prev[v] = u

else if alt < dist2[v]

dist2[v] = alt

prev2[v] = u

return (dist2[],prev2[]);

So this algorithm can be run a single time and find the second shortest path, if your goal is to find both the first and second shortest paths and return them simultaneously, the algorithm can be run twice or more data structures can be implemented at the cost of space to implement the return of multiple sets of data. The algorithm has a worst case scenario of running in O((E + V)log(V)) time, and an average case of O(E+Vlog(E/V)log(V)). This is taking into account grid and array access costs which are O(1) and O(log(V) respectively. This adjusted algorithm will return correct results, and these arrays can be traversed in conjunction with the grid to mark a path through the grid which can be visualized. Because the adjustment to the algorithm aims to make the smallest miss-step from the source to the destination, there are cases where the second shortest path may not be entirely accurate—this is acceptable, for in a practical and applied approach to the problem the difference between our recorded second shortest path and the correct SSP will be negligible.

**Adjusted Dynamic-Programming Method**

We take a similar approach with the dynamic programming method, this time, since we are replacing values within the grid itself, we must also use another grid of the same size to store previous node data—this is how we track the shortest, and second shortest paths. We must add in the additional cost of a grid of the same size to our space complexity, as well as a few vector and stack containers to hold our paths. Again, if we want to return both the first and second shortest path in one run additional data structures will be required to perform this, but the algorithm will simply print out both the first and second shortest paths for the sake of simplicity.

int grid\_SP(int cost[R][C], int m, int n)

int i, j <-- 1

stack first\_path[]

stack ssp[]

int back\_track[]

int min\_diff[]

xypair prev[R][C]

int total\_cost[R][C]

total\_cost[0][0] <-- cost[0][0]

prev[0][0].x <-- 0

prev[0][0].y <-- 0

for j from 1 to n do:

total\_cost[0][j] <-- total\_cost[0][j - 1] + cost[0][j]

prev[0][j].x <-- 0

prev[0][j].y <-- j - 1

end for;

for i from 1 to n do:

total\_cost[i][0] <-- total\_cost[i - 1][0] + cost[i][0]

prev[i][0].x <-- i - 1

prev[i][0].y <-- 0

for j from 1 to n do:

total\_cost[i][j] <-- min(total\_cost[i - 1][j], total\_cost[i][j - 1]) + cost[i][j]

if (min(total\_cost[i - 1][j], total\_cost[i][j - 1]) == total\_cost[i - 1][j])

xypair p

p.x <-- i - 1

p.y <-- j

prev[i][j] <-- p

else //previous node matrix op

xypair p

p.x <-- i

p.y <-- j - 1

prev[i][j] <-- p

end-for

end-for

//end shortest route code

vector back\_track[] //keep backwards path for simplicity in backtracking

j = C

for i from R to 0 do: //find and print path in 1's and 0's

if (prev[i - 1][j - 1].x == (i - 2) && prev[i - 1][j - 1].y == (j - 1))

first\_path[] <-- 0 //inserting 0 into stack, then into array for simplifying iterations

back\_track[] <-- 0

else if (prev[i - 1][j - 1].y == (j - 2) && prev[i - 1][j - 1].x == (i - 1))

first\_path[] <-- 1

back\_track[] <-- 1

j--

i++

end-for

///begin second shortest path code

for i from 0 to size of min\_diff[]

min\_diff[i]<--10000

end-for

int sml, grt, diff

i <-- R - 1

j <-- C - 1

for k from 0 to size of back\_track[]

if (back\_track[k] == 0)

grt <-- total\_cost[i][j - 1]

sml <-- total\_cost[i - 1][j]

diff <-- abs(grt - sml)

min\_diff[k] <-- diff

if (i != 0)

i--

else if (back\_track[k] == 1)

grt <-- total\_cost[i - 1][j]

sml <-- total\_cost[i][j - 1]

diff <-- abs(grt - sml)

min\_diff[k] <-- diff

if (j != 0)

j--

end-for

int min <-- 10000

int min\_index

for k from 0 to size of back\_track[] //finding minimum diff and the index for pathfinding

if (min\_diff[k] < min)

min = min\_diff[k]

end-for

for k from 0 to size of back\_track[]

if (min\_diff[k] == min)

min\_index = k

end-for

for k from 0 to min\_index //populating second path stack up to turning point

ssp[] <--back\_track[k]

end-for

i = R

j = C

for k from 0 to min\_index //begin keeping track of location while iterating through path up to turning point

if (back\_track[k] == 1)

j--

if (back\_track[k] == 0)

i--

end-for

if (back\_track[min\_index] == 0) //check shortest path, choose opposite at min\_index

ssp[] <-- 1

j--

else

ssp[] <-- 0

i--

for k from i to 0 do: //following previous nodes back to origin

if (prev[k - 1][j - 1].x == (k - 2) && prev[k - 1][j - 1].y == (j - 1))

ssp[] <-- 0

else if (prev[k - 1][j - 1].y == (j - 2) && prev[k - 1][j - 1].x == (k - 1))

ssp[] <-- 1

j--

k++

end-for

print( "First shortest path (1=right, 0=down): " )

while (first\_path.size != 0)

print( first\_path.top + " " )

first\_path.pop

print("Second shortest path (1=right, 0=down): " )

while (ssp.size != 0){

print( ssp.top + " " )

ssp.pop

return total\_cost[m - 1][n - 1]

This algorithm proves to be much more efficient, due to the nature of our data and the specific constraints on the path we must take. We can exploit the grid’s structure to make our calculations easier, and have an average case of O(n) time, where n is the number of vertices—to arrive at this conclusion we must break down our algorithm and estimate the costs—the building of our total cost array will cost m\* n, or V, and various other calculations such as finding the path and storing this information only increase this cost by a constant amount. The total complexity can be determined to be: (m\*n) + n + 9C, where C is the constant cost of the path constructed by the algorithm. This is clearly better, given a basic case O(n) < O(nlog(n)). The dynamic programming approach uses the inherent structure to produce desired results in less time than Dijkstra’s implementation. Because the algorithm uses the first shortest path to determine the second shortest path, there is a chance that the second shortest path may not be accurate in edge cases. This is okay, because the function works to find a second shortest path regardless, and if it is not the true second shortest path, it is very close.

**Comparing adjusted algorithms**

As it has already been stated, the dynamic programming approach has a smaller time complexity, even in the case of a grid that is 10000 by 10000. The time complexity O(n) outperforms Dijkstra’s even in the best case for Dijkstra, of O(E log(n)), and this is proved in the timing study. Also, the dynamic programming approach is more readable, and easier to conceptualize—therefore it is an easier task to implement this algorithm in multiple languages. For much smaller graphs Dijkstra’s adjusted algorithm will outperform the dynamic programming approach, but it is the purpose of this project to study grids asymptotically bounded by 10000, thus the dynamic programming option will be implemented.

**Implementation**

The algorithm will be implemented in C++, the main reason being resource management capabilities of the language, as this program may be strained by large graphs, resource management efficiency is important. Using the dynamic approach, we have several sets of data that we are manipulating throughout the function, so managing these resources is significant. Using Djikstra’s approach, when implementing the heap as a set, it is important to note that set objects are comprised of <vertex, shortest distance to vertex> pairs, and these are compared both by vertex number and value—we are essentially establishing a mapping of vertex number to the index of its distance cost. This is important to maintain the heapness of the priority queue, as these objects continue to be compared by distance, but can be accessed by vertex number—the vertex number isn’t provided as part of the graph, and must be created in some way. Ultimately the dynamic programming method is used, as the ‘grid’ nature of the graphs makes an iterative approach favorable as far as time complexity goes, and we don’t have to worry about maintaining a heap structure

**Timing Study**

When we consider the largest scenario for our grid, m is equal to 10,000 and n is equal to 10,000, thus there are 100,000,000 nodes in our graph. We can use these values to compute how our algorithms will perform at the boundedness described in our project. For Dijkstra’s algorithm, we compute (VNlog(V))—where N is the number of edges attached to each node, which is 1,600,000,000 k&b operations, and being generous assuming each node only has 2 edges attached to it. If we look at the dynamic programming approach, we see a large difference in time cost. The dynamic programming approach has a time complexity of T(n) = (m\*n) + n + 9C where C is roughly equal to (m\*n)/2. Plugging in some numbers gives us a result of 550,010,000 k&b operations at the maximum input our project requires—this runs in about a third of the time of Dijkstra’s, without putting too much stress on space required to hold the data. If the input size were to double, this would only be more in favor of the dynamic programming method, thanks to the nature of the grid and the way data is provided so that this problem can be solved iteratively and simply.

**Epilogue**

In reflection, I believe the nature of the project almost implied using an adjacency matrix to represent our problem, and the use of it is integral to the efficiency of the solution brought forth. Because we can refer to all of these graphs as a grid with less than 4 edges, the dynamic programming approach was the easy choice for implementing a solution to the second shortest path problem. I have learned that when presented with a new unique problem, one must first really make sure they understand the problem conceptually, then begin to work on ideas of how to solve the problem. After that, one must test and challenge these ideas by attempting to implement them—this is where the real breakthroughs happen, and you are forced to break down and solve smaller problems to work towards your main goal. If I were to do this project again, I would much like to work with others as a team; I feel more ideas will be brought to the table, and matters of efficiency and complexity are easier resolved with multiple perspectives. I have learned from this to test my thoughts early and often, and challenge my conceptual understanding of a problem so that I may be more aware of how to solve it efficiently. For future students, I would give the same advice that I have learned—test your ideas, implement them into something that you can see results from, then go back to the drawing board and refine your work. With new problems that seem to have multiple solutions, it’s important to think about what tools are right for the job, and sometimes you need to just pick a tool and use it and measure results; this can get you in the right mindset for what needs to be accomplished, and what route is the best to take to get there.’

**Appendix A:**

Source files were compiled and run through Visual Studio 2013

Machine: Hewlett-Packard 64-bit Operating System Windows 8

Files included are as listed:

Source.cpp – includes main function, calls dp\_ssp function

Dp\_method.h – our dynamic programming approach to finding the second shortest path

D4.h –approach to finding second shortest path using dijkstra’s algorithm

**Appendix B:**

Path Output

First example will show original test matrix, total cost matrix, first path, second path, and total cost to travel

Custom Grid -- 4 x 4

Total cost matrix:

1 6 16 32

26 11 31 41

46 21 56 86

66 56 106 156

Previous node matrix:

(0,0) (0,0) (0,1) (0,2)

(0,0) (0,1) (1,1) (1,2)

(1,0) (1,1) (2,1) (1,3)

(2,0) (2,1) (2,2) (2,3)

First shortest path (1=right, 0=down):

1 0 1 1 0 0

Second shortest path (1=right, 0=down):

1 1 1 0 0 0

First Path Travel Cost: 156

Second Path Travel Cost: 157

First and second shortest paths for input files provided--

grid1

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 288

Second Path Travel Cost: 292

grid2

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 2690

Second Path Travel Cost: 2699

grid10

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 2012

Second Path Travel Cost: 2015

grid11

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 2252

Second Path Travel Cost: 2263

grid12

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 2519

Second Path Travel Cost: 2519

grid13

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 3603

Second Path Travel Cost: 3603

grid14

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 3900

Second Path Travel Cost: 3901

grid15

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 2871

Second Path Travel Cost: 2871

grid16

grid17

grid18

grid19

grid20

grid21

First shortest path:

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

Second shortest path:

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

First Path Travel Cost: 141025

Second Path Travel Cost: 141025

**Works Cited/Helpful Resources**

Miguel Oliveira - <http://paginas.fe.up.pt/~ei05053/doku.php?id=graphs:secondshortestpath>

Wiki - <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

Wiki - <https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm>

<http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>

<http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part2.pdf>