Julius D. Higiro
13 Nov 2015

Dijkstra's Algorithm Analysis

1. Formulation of problem:

The algorithm was defined to solve the problem of finding the shortest path from a single source vertex to every other vertex in the graph. The algorithm takes as input a graph G = (V, E) where V is the set of vertices in the graph, and E is the set of edges between the vertices with a non-negative weight. The algorithm returns as output the shortest path from the source vertex to every other vertex in the graph. The algorithm is a greedy algorithm that can be used on directed or undirected graphs.

2. Pseudocode for Dijkstra's algorithm:

Dijkstra(G, start_node)
// graph G = (V, E), edge (u,v)$\in E$, v$\in V$
// start_node is the source vertex
initialize all entries of an array *distance* of length V to infinity
distance[start_node] ← 0
priority queue PQ← contains every vertex in the graph G
S ← ∅
While PQ is not empty:
    u ← PQ.removeMin()
    S ∪ $\{u\}$ add u to the list of explored verteces
    for each vertex v that is a neighbor of u:
Lines 1: if *distance*[u]+weight of edge(u,v) < *distance*[v]:
Lines 2: *distance*[v] ← *distance*[u]+weight of edge(u,v)

update v in PQ with *distance*[v]
return *distance*

Proof of Correctness:

Claim: For the set S which contains the set of nodes that have been explored, at any stage of Dijkstra's algorithm, the path for each node in S to the starting node is the shortest path distance.

Base case:

Let S = 1, then S = {s}, s is the only node in S and with the distance[s] = 0 = length of the empty path from s to s. We have found the shortest path and so the above claim is trivially true.

Inductive hypothesis:

Suppose our claim holds true for |S| = K, where K>=1.
Let the number of verteces in S increase to K+1.
Let v be the next node that is added to S to increase it to K+1 with the edge (u,v).
Let Pv represent the path from s to v.
Let x,y be the first edge leaving the set S with x in the set S.

By I.H., it follows that Pu is the shortest path from s to u for every u that is in our set S. Now, if we consider an alternative path from s to v marked by P that goes through x and y, it turns out that the path P will have been longer than Pv by the time it leaves the set S. Per lines 1 and 2 of the pseudocode (i.e. the relaxation operation), Dijkstra's algorithm at K+1 iteration will have looked at node y through x and node v through u and it will have added the path with the minimal cost. Since the algorithm chose to add v, it means there is no path from s to y that goes through x that is shorter than the path Pv. The path Pv chosen by the algorithm is less than or equal to the alternative path and there is no shorter path.

4. Complexity of Algorithm:

Dijkstra's algorithm has a time complexity of $O(V^2 + E) = O(V^2)$ when an array or list is used to keep track of the cost of the vertices. If we consider that we have an unsorted array, it takes a time complexity of O(V) to perform the operation of extracting the vertex with the minimum distance value inside the while loop. The reason is the whole array has to be searched in order to find and extract the vertex with the minimum cost. Since there exists V operations, the total time is $O(V^2)$. The inner for loop iterates E times with constant time per iteration. So the algorithm complexity is $O(V^2 + E) = O(V^2)$. The structure of the graph (adjaceny matrix or adjacency list) doesn't change the complexity of the algorithm. The time complexity of $O(V^2)$ remains the same with both graph structures when a list or an array is used.

However, the algorithm has a time complexity of O((E+V)lgV) = O(ElgV) when a heap/priority queue is used to keep track of the cost of the vertices and when an an adjaceny list is the input graph. It takes a time complexity of O(lgV) to perform the operation of extracting the vertex with the minimum distance value. Since there exists V operations, the total time for building the heap is O(lgV). The operation *decrease()* that is used to restrict the number of verteces in the heap is performed in the for loop in a time of O(lgV) and there exitst at most E operations. So the total time complexity of the algorithm is O(ElgV). It should be noted that using an adjacency matrix as the graph data strucuture in the case where a heap/priority queue is used, results in a time complexity of $O(V^2\text{+VlgV}) = O(V^2)$. Also, the time complexity of the algorithm can be improved further with the use of a fibonnaci heap that results in an overall time complexity of O(E+VlgV). Lastly, the space complexity for Djikstra's algorithm is O(V+E).

3. Implementation of Dijkstra's algorithm in Python:

```
import numpy as np

def dijkstra(graph, source):

    # Matrix size
    n = len(graph)

    # Assign an intital distance value of infinity to all the
```

```
    # nodes
    infinity = float('inf')
    distance = [infinity for i in range(n)]
    # Assign a distance value of zero to the starting node
    distance[source] = 0
    current_node = source
    # Variables to store visited nodes and their distance from
    # the starting node
    visited_nodes = list()
    visited_nodes_and_cost = list()
    # Variable to store nodes and their distance from the
    # starting node
    dictionary = {}
    dictionary.update({source:distance[source]})
    # Variable to store the previous nodes, reversed the
    # assignment of key,value
    # in order to correctly update
    preceeding_nodes = {}

    while((len(dictionary))!=0):

        # Identify the node with the smallest distance value in
         # the dictionary
        current_node, cost = min(dictionary.items(), key=lambda
x:x[1])
        # remove the node with the smallest distance value from
         # the dictionary
        del dictionary[min(dictionary, key=dictionary.get)]
        # mark the current node as visited
        visited_nodes.append(current_node)
        visited_nodes_and_cost.append((current_node,cost))

        # check distances from all neighboring nodes to the
        # current node
        for j in range(n):
            # only evaluate nodes in the graph that contain edges
            # and that have not been visited
            if(graph[current_node][j]!=0 and j not in
visited_nodes):
                # update the distances from neighboring nodes to
                # the current nodes
                # and store in the dictionary
                if(distance[current_node] +
graph[current_node][j] < distance[j]):
                    distance[j] = distance[current_node] +
graph[current_node][j]
                    dictionary.update({j:distance[j]})
                    preceeding_nodes[j] = current_node

    return visited_nodes_and_cost, preceeding_nodes
```

```
def main():

      # prompt user to enter the file name
      filename = raw_input('Enter the file name: ')+'.txt'

      # load adjacency matrix from text tile
      graph = np.loadtxt(filename)

      # prompt user to enter the source node
      start_node = input('Enter starting node: ')
      while start_node>=len(graph) or start_node<0:
         start_node = input('The value entered is out of bounds.
Try again: ')

      # prompt user to enter the source node
      end_node = input('Enter end node: ')
      while end_node>=len(graph) or start_node<0:
         end_node = input('The value entered is out of bounds. Try
again: ')
      target_node = end_node

      costs = []
      previous = {}
      costs, previous = dijkstra(graph, start_node)

      # print the shortest path from the start node to every other
      # node
      print "The shortest path from the start node %d to every
other node in the graph is \n %s" %(start_node,costs)

      # find the shortest path from source node to target node
      SSSPath = []
      while True:
          SSSPath.append(end_node)
          if(end_node == start_node):
              break
          end_node = previous[end_node]
      SSSPath.reverse()

      # print the shortest path from the start node to the target
      # node
      print "The shortest path from the start node %d to the
target node %d is %s with a distance of %s"\
      %(start_node, target_node, SSSPath,
dict(costs).get(target_node))

main()
```