DIJKSTRA'S SHORTEST PATH ALGORITHM

REMY OCHEI

Abstract. And introduction to Dijkstra's Shortest Path Algorithm

Hello dear student! This is an implementation of Dijkstra's Shortest Path algorithm. Dijkstra's original algorithm finds the shortest path between two nodes in a graph. A common variant of Dijkstra's algorithm fixes one node as a source, and finds the shortest path to all other nodes.

You might find yourself wondering, what's a graph, what's an algorithm, what's a Dijkstra? In that case, I have included the the wikipedia introductions for those concepts below. The wikipedia pages themselves serve as excellent places to start further inquiries into those topics. It's my goal that you should be able to follow along here without visiting those pages, but they might help to resolve an unforeseen point of confusion for you. If you have some familiarity with these concepts already, feel free to skip the next section.

1. Introductions

1.1. **Dijkstra's Algorithm.** Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

1.2. **Graph.** In mathematics, and more specifically in graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called link or line). Typically, a graph is depicted in diagrammatic form as a set of dots or circles for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.
https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

1.3. **Algorithm.** In mathematics and computer science, an algorithm is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, automated reasoning, and other tasks.
https://en.wikipedia.org/wiki/Algorithm

2. Algorithm and Proof of Correctness

2.1. **Definitions.** Mathematically, a **graph** G is an ordered pair (V,E), where V is a set of nodes or vertices and E is a set of edges. The elements of E are either ordered (x,y) or unordered pairs {x,y}, where x and y are elements of V. If the elements of E are unordered pairs, then G is said to be undirected. In a directed

graph, the elements of E are ordered pairs and a distinction is made between an edge from x to y (x, y) and an edge from y to x (y, x). In a directed graph, the edges are said to have an orientation.

If we assign a weight w to each edge, then we have a **weighted graph**. In practical applications a weight might be taken to represent a length, cost, or capacity. Dijkstra's algorithm, first published in "A Note on Two Problems in Connexion with Graphs" (Dijkstra 1959), finds the shortest path between two nodes of a weighted graph, where the weights are taken to be distances (and are therefore non-negative). Since an undirected graph can be thought of as a directed graph where the edges (x, y) and (y, x) have the same weight (for all x and y), it is sufficient to consider directed graphs only, without loss of generality.

A **path** can be unambiguously represented as a sequence of edges or a sequence of vertices.

Breadth-first search is an algorithm for exploring the vertices of a graph. It starts at a given vertex (often called the root), and explores first all its neighbors before moving on to the neighbors of the neighbors, and the neighbors of the neighbor's neighbors etc., until all the vertices of the graph have been explored. In other words, first the root is visited, then all vertices one edge away from the root are visited, then two away etc., until all vertices are explored.

Breadth-first search can be contrasted with **depth-first search**, in which exploration proceeds by traveling as far as possible from the root until no new vertices can be found. At that point the algorithm backtracks until it can find an unvisited vertex. Once found, it proceeds forward again. This process repeats until the entire graph is explored.

Lemma 1. *If r is a vertex on the shortest path from p to q, SHORTEST(p, q), then SHORTEST(p, q) = SHORTEST(p, r) + SHORTEST(r, q)*

Lemma 1 offers as a *recursive* definition for the shortest path.

At a high level, Dijkstra's algorithm for finding the shortest path between the vertices p and q in graph G performs a **breadth-first search** of G starting from p. The first time any vertex r is visited, the set of its neighbors N is considered. For each vertex v in N, if length[TENTATIVE(p,r) + {(r,v)}] < length[TENTATIVE(p,v)], then the old path is replaced, and TENTATIVE(p,v) is set to TENTATIVE(p,r) + {(r,v)}.

Lemma 2. *After every visit in Dijkstra's algorithm, the shortest of all tentative paths out of p is known to be a shortest path.*

In this case the path is no longer considered tentative, and we add the endpoint of this path to the set of vertexes for which the shortest path is known. We might say that Dijkstra's algorithm *greedily* chooses among the tentative paths after each original visit to a vertex.

Proof (Via Contradiction). Let TENTATIVE(p,r) be the shortest of all tentative paths. Assume there is some other path between p and r that is shorter than TENTATIVE(p,r). Then this path must go through some vertex s, s != r, that is not on TENTATIVE(p,r) such that length[TENTATIVE(p,s)] < length[TENTATIVE(p,r)] and length[TENTATIVE(p,s) + {(s,r)}] < length[TENTATIVE(p,r)], violating our assumption that TENTATIVE(p,r) is the shortest of all tentative paths. QED.

When the shortest tentative path ends at q, then SHORTEST(p,q) is known, and the algorithm terminates.

3. Implementation

The first step is reading and parsing the text file where the graph is stored in order to construct our graph object. There are three different representations for storing a graph, adjacency lists, adjacency matrices, and incidence matrices. Of these, the incidence matrix does not have an intuitive way to represent the lengths of edges, so we ignore that format (however, an example incidence matrix and code to parse it is included for the interested reader).

This first piece of code is to parse and comprehend the input and relies on the argparse module. We take in four inputs, the input file, the format of that file, the starting vertex, and the ending vertex. Examples of valid input files are included in the project folder.

```
import argparse
import operator

# The argparse module helps to parse command line arguments

parser = argparse.ArgumentParser(description="Execute Dijkstra's
    ↪ Algorithm on an input file")
parser.add_argument('inputFile', metavar='I', help='file that
    ↪ contains the representation of the graph')
parser.add_argument('graphFormat', metavar='F', help='either
    ↪ adj_list or adj_matrix')# or inc_matrix')
parser.add_argument('startVertex', metavar='p', help='the name of
    ↪ the starting vertex')
parser.add_argument('endVertex', metavar='q', help='the name of the
    ↪ ending vertex')
args = parser.parse_args()
```

Next, we define the Node object, the Node object being our internal representation of the vertices of the graph

```
# Although our program can read in 2 formats for the representation
    ↪ of the graph, internally we will use an adjacency list.

class Node:
    def __init__(self, name):
        self.name = name
        self.neighbors = dict()

    def addNeighbor(self, v, d):
        self.neighbors[v] = d

    def toString(self):
        return self.name
```

In this implementation, the Node object carries within it references to its neighbors, mirroring the structure of an adjacency list.

The next block of code handles the case where the input file is an adjacency list.

```
if args.graphFormat == 'adj_list':
    for line in f:
        splitLine = line.split(':')
        nodeName = splitLine[0]
        if nodeName not in nodes:
            nodes[nodeName] = Node(nodeName)
        nodeNeighbors = splitLine[1].split()
        for x in nodeNeighbors:
            v,d = x.split(',')
            if v not in nodes:
                nodes[v] = Node(v)
            nodes[nodeName].addNeighbor(nodes[v], float(d))
```

Next, we handle the adjacency matrix case.

```
if args.graphFormat == 'adj_matrix':
    firstLine = True
    nodeNames = []
    for line in f:
        if firstLine:
            nodeNames = line.split()
            for x in nodeNames:
                nodes[x] = Node(x)
            firstLine = False
        else:
            splitLine = line.split()
            nodeName = splitLine[0]
            splitLine = splitLine[1:]
            for i,d in enumerate(splitLine):
                d = float(d)
                if d:
                        nodes[nodeName].addNeighbor(nodes[nodeNames[i]],
                            ↪ d)
```

When storing the shortest path between p and r, it is sufficient to merely associate r with the last edge or penultimate vertex on the path.

Let (s, r) be the last edge of the path SHORTEST(p, r). Then, according to lemma 1, SHORTEST(p,r) = SHORTEST(p, s) + {(s, r)}. If we have this association for all vertices of the graph, then we can trace backwards from any endpoint to p to reconstruct the shortest path. The code for doing so is at the very end.

```
v = q NumerischeMathematlk
path = []
while v != p:
    path.append(v)
    v = shortest[v][0]
path.append(p)
for v in reversed(path):
    print v.toString(),
```

```
    if v is not q:
        print ' --> ',
```

Lastly, let's look at the code for the algorithm itself. We create two dictionaries, tentative and shortest, which mirror one another in their structure. Each dictionary takes as a key some vertex r and associates it with the tuple (vertex v, length d). In tentative, v is the penultimate vertex on the shortest *known* path to the ending vertex q. Shortest is where the path is stored once the algorithm is confident that a tentative path is truly the shortest path between the starting vertex p and the vertex r.

```
startName = args.startVertex
endName = args.endVertex

p = nodes[startName]
q = nodes[endName]

# tentative[r] is of the format (v, d), where v is the penultimate
    ↪ vertex on the tentative path between p and r, and d is the
    ↪ length of that path.

tentative = dict()
for key in nodes:
    tentative[nodes[key]] = (p, float('inf'))

# shortest[r] is of the format (v, d), where v is is the penultimate
    ↪  vertex on the shortest path between p and r, and d is the
    ↪ length of that path.

shortest = dict()
shortest[p] = (p, 0) # we initialize shortest to only contain the
    ↪ starting node
```

A tentative path becomes a shortest path at the end of an iteration of the algorithm, where the *shortest tentative path* of all tentative paths (to any node, not just the ending node) is taken to be an actual shortest path.

```
while q not in shortest:
    for v in shortest:
        for r in v.neighbors:
            if tentative[r][1] > shortest[v][1] + v.neighbors[r]:
                tentative[r] = (v, shortest[v][1] + v.neighbors[r])
    # sort all tentative paths
    tent = [(key, tentative[key][1]) for key in tentative if key not
        ↪ in shortest]
    tent.sort(key=operator.itemgetter(1))

    r = tent[0][0]
    shortest[r] = (tentative[r][0], tentative[r][1])
```

And that's Dijkstra's algorithm. Next, we'll take a look at the Bellman-Ford and Floyd-Warshall algorithms, alternates for finding the shortest path between two vertices.

References

[1] E. W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik (1959).

Of No School Nor Style, 1219 Bethel School CT, Coppell, TX 75019
Email address: do.infinitely@gmail.com