Dijkstra algorithm assignment
Optimization

Nieves Montes Gómez (1393150)

November 2019

12. After graduating you accept a job with Aerophobes-R-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

Suppose one of your customers wants to fly from city X to city Y . Describe an algorithm to find a sequence of flights that minimizes the total time in transit—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights.

With all our data, we start by building a directed weighted graph. Each node represents an airport: $V = \{X, A, B, ..., Y\}$, where X is the source node (the start of the trip), and Y is the destination node. $A, B, ...$ are all other airports. In this graph, there is an edge from airport U to V only if there is a direct flight between the two.

We must maintain a *now* variable with the current time. This information needs to be stored because we cannot catch a flight whose departure time is earlier than the current time. Depending on the value of this variable, flights whose departure time is earlier than *now* cannot be considered.

From a given origin airport u and time, we scan all possible successors (i. e. all those with an direct from u to v, that have not departed yet) by assigning the following weight to the edge $w(u \to v)$:

$$\begin{aligned} w(u \to v) =& (\text{waiting time at u before flight to v departs}) + \\ &+ (\text{flight time from u to v}) = \\ &= \text{arrival time at } v \end{aligned}$$

In case there is more than one flight available between the two locations, we select the one that generates the lowest weight for the edge.

Now, we can apply the Dijkstra algorithm to this graph. The only difference to the usual version is that the *now* variable must be updated after we reach a new node with the value of the arrival time of the flight we took to get there. The algorithm then proceeds as displayed in the following page. Each node has a *cost* function associated to it, which stores the (provisional) minimum time it takes to get there.

```
cost(origin) ← 0;
for all airports (v) other than source do
    cost(v) ← ∞;
end
Q ← set of all airports;
while (Q ≠ ∅) do
    u ← the airport with lowest cost(u), among those in Q;
    if u ≠ source then
        now ← arrival time of flight from previous airport;
    end
    if u = destination then
        Congratulations! You have reached your destination;
        break;
    end
    for all airports (v) with direct flight from u, v still in Q, and departure of flight u → v later
     than now do
        w ← min_{flights u→v}(waiting time + air time) = min_{flights u→v}(arrival time to v);
        if cost(v) > cost(u) + w then
            cost(v) ← cost(u) + w;
            pred(v) ← u;
        end
    end
    delete u from Q;
end
```

In order to implement this algorithm, we load some data of 2013 US domestic flights departing from NYC airports that can be found here. After cleaning the data, we have access to departure and arrival dates of over 325,000 flights between 107 different airports. The data only includes flights that depart from three different airports, so we artificially shuffle airports to generate flights between many locations. A Python program that implements this algorithm and outputs a .txt file with the computed route can be found here. Source and destination airports are chosen randomly. Following are some examples of calculated routes:

```
ROUTE FROM TVC TO BUR
--------------
973,RSW,BUR,2013-01-02 07:44:00,2013-01-02 10:17:00
--------------
800,ROC,RSW,2013-01-01 21:15:00,2013-01-01 22:40:00
--------------
609,SDF,ROC,2013-01-01 17:32:00,2013-01-01 20:28:00
--------------
412,MTJ,SDF,2013-01-01 14:30:00,2013-01-01 17:18:00
--------------
177,TVC,MTJ,2013-01-01 09:08:00,2013-01-01 12:28:00
--------------
```

```
ROUTE FROM EWR TO IAD
--------------
954,BNA,IAD,2013-01-02 07:29:00,2013-01-02 08:46:00
--------------
827,RDU,BNA,2013-01-01 23:06:00,2013-01-02 00:28:00
--------------
661,JFK,RDU,2013-01-01 18:27:00,2013-01-01 21:05:00
--------------
480,CAE,JFK,2013-01-01 15:29:00,2013-01-01 17:33:00
--------------
283,EWR,CAE,2013-01-01 11:33:00,2013-01-01 14:48:00
--------------



ROUTE FROM SLC TO PWM
--------------
1120,MDW,PWM,2013-01-02 10:03:00,2013-01-02 11:05:00
--------------
536,RIC,MDW,2013-01-01 16:20:00,2013-01-01 19:45:00
--------------
151,SLC,RIC,2013-01-01 08:48:00,2013-01-01 10:01:00
--------------



ROUTE FROM MSN TO PWM
--------------
1013,GSP,PWM,2013-01-02 08:20:00,2013-01-02 11:17:00
--------------
753,RSW,GSP,2013-01-01 20:15:00,2013-01-01 21:49:00
--------------
581,PDX,RSW,2013-01-01 17:05:00,2013-01-01 19:24:00
--------------
207,MSN,PDX,2013-01-01 09:37:00,2013-01-01 11:27:00
--------------
```

13. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road won't be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Arizona. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph G = (V, E), where every edge e has an independent safety probability p(e). The safety of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.

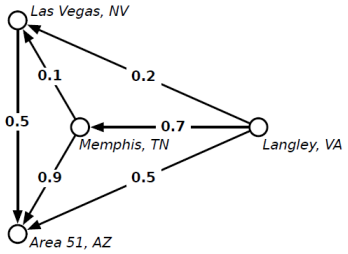

Figure 1: Directed graph with edge probabilities.

We start by assigning a safety function saf(v) to all nodes in the graph. This function corresponds to the probability of not being abducted while following a path from the source node to any other node. Th algorithm starts by assigning safety 1 to the source node (you cannot be abducted by going to the source node given that you are already in the source node) and 0 to all other nodes.

The algorithm then proceeds by extracting the node with the highest safety function among all those in a queue. It scans its successors and computes their safety function if they were to be reached from the current node. This computation is performed assuming that the probability of non-abduction is independent among all the roads in the graph. Hence, the probability of getting safe to node v from node u is computed as $saf(u) \cdot p(u \to v)$, where $p(u \to v)$ is the probability of having a safe trip from u to v. These probabilities are indicated in Figure 1.

So, if the probability of reaching a successor safe form the current node is *higher* than the established value, the safety function of the successor is updated and its predecessor is set to be the current node. Notice that, unlike the typical Dijkstra algorithm, in this case we want to maximize the safety function at each node. The pseudocode for the algorithm is in the following page.

A C++ program that implements this algorithm for this particular problem can be found here. The output it produces is:

```
Congratulations, you have reached area 51, AZ.
This has been your trip:
Position: VA
        Safety function: 1
Position: TN
        Safety function: 0.7
Position: AZ
        Safety function: 0.63
```

```
1  saf(source) = 1;
2  pred(source) = NULL;
3  for all nodes other than the source do
4  |  saf(node) = 0;
5  |  pred(node) = NULL;
6  end
7  S ← ∅;
8  Q ← set of all nodes including the source;
9  while Q is not empty do
10 |  currentNode ← node with highest saf(v) among those in Q;
11 |  if saf(currentNode) = 0 then
12 |  |  Sorry, you have probably been abducted by aliens;
13 |  |  break;
14 |  end
15 |  if currentNode = Destination then
16 |  |  Congratulations, you have made it to your destination;
17 |  |  break;
18 |  end
19 |  for all successors of currentNode that are still in Q do
20 |  |  succCurrentSafety ← saf(currentNode) × p(currentNode → succ);
21 |  |  if saf(succ) < succCurrentSafety then
22 |  |  |  saf(succ) ← successorCurrentSafety;
23 |  |  |  pred(succ) ← currentNode;
24 |  |  end
25 |  end
26 |  add currentNode to S;
27 |  remove currentNode from Q;
28 end
```

14. (...) Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total expected number of vampires encountered along the path is as small as possible. Be sure to account for both the vertex probabilities and the edge probabilities!

Let's assume that the events of finding a vampire along any of the nodes or edges are independent. Let's say that, starting at the tent node, we take a first step to node v_1 through edge e_1. The probability of finding a vampire in this subpath (SP) is:

$$p(SP_1) = p(e_1 \cup v_1) = p(e_1) + p(v_1) - p(v_1) \cdot p(e_1)$$

Now, we take the next step to node v_2 through edge e_2. Considering the independence of finding vampires in different nodes and/or edges, the probability of finding a vampire along the total subpath, up to this point, is:

$$p(SP_2) = p(SP_1 \cup e_2 \cup v_2) = p(SP_1) + p(e_2 \cup v_2) - p(SP_1) \cdot p(e_2 \cup v_2) =$$
$$= p(SP_1) + (p(e_2) + p(v_2) - p(e_2) \cdot p(v_2)) - p(SP_1) \cdot (p(e_2) + p(v_2) - p(e_2) \cdot p(v_2))$$

In general, when scanning possible next nodes to follow in our path, the probability of meeting a vampire once we get there is updated through this expression:

$$p(SP_i) = p(SP_{i-1}) + (p(e_i) + p(v_i) - p(e_i) \cdot p(v_i)) - p(SP_{i-1}) \cdot (p(e_i) + p(v_i) - p(e_i) \cdot p(v_i))$$

Our algorithm assigns a risk function to each node $\mathsf{risk(v)}$, which assesses the probability of meeting a vampire for a subpath that starts at the tent node and ends in v. Note that the probabilities of encountering a vampire in the nodes or edges are *parameters* of the problem, and hence are not updated as the algorithm proceeds. The defined risk function, however, is updated. The Dijkstra algorithm applied to this situation finds a path to each node that minimizes it. It starts by assigning $risk(source) = p(source)$ and $risk(v \neq source) = 1$. Then, it selects the node with the lowest risk that have not been visited. Next, it computes the risk of all adjacent nodes if the path to be taken was to be from the current node. It compares it with the current risk of the predecessors, and if it finds it to be lower for the new path, it updates the risk of the predecessor to the new lower value and its predecessor to be the current node. It repeats the process until an exit node is found. The pseudocode for this algorithm is displayed in the next page.

```
risk(source) = p(source);
for all nodes other than the source do
    risk(node) = 1;
end
Q ← set of all nodes including the source;
while Q is not empty do
    u ← node with lowest risk(v) among those in Q;
    advance to u;
    if risk(u) ≥ 1 then
        Sorry, you are dead;
        break;
    end
    if u = EXIT then
        Congratulations, you have made it out alive;
        break;
    end
    for all v successors of u that are still in Q do
        aux ← p(u → v) + p(v) − p(u → v) · p(v);
        if risk(v) > risk(u) + aux − risk(u) · aux then
            risk(v) ← risk(u) + aux − risk(u) · aux;
        end
    end
    remove u from Q;
end
```

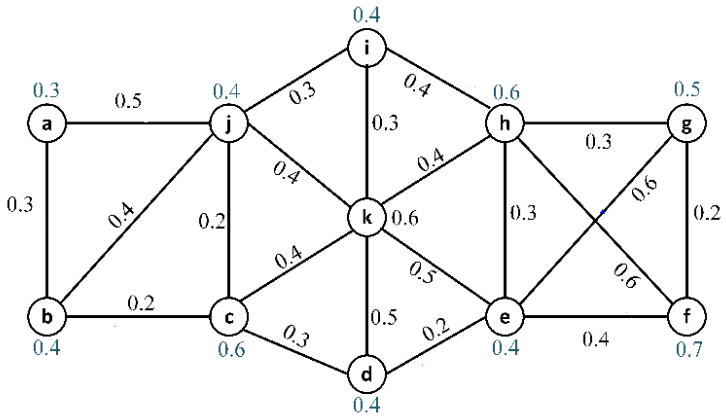

Figure 2: Weighted graph that emulates the map of the park. The starting node (tent) is a. Exits is located at node g.

In order to implement the algorithm in an actual graph, we simulate that the map of the national park can be modeled by the graph displayed in Figure 2, where the starting node (the tent) is a and the only possible exit is located at node g. The program that runs the algorithm can be found here. The output produced by the program is:

```
Congratulations, you have made it out alive.
This has been your trip:
```

```
Position: a
        Risk function: 0.3
Position: b
        Risk function: 0.706
Position: j
        Risk function: 0.79
Position: c
        Risk function: 0.90592
Position: i
        Risk function: 0.9118
Position: k
        Risk function: 0.9496
Position: d
        Risk function: 0.960486
Position: h
        Risk function: 0.978832
Position: e
        Risk function: 0.981033
Position: g
        Risk function: 0.992591
```