ALGORITHMS AND DATA-STRUCTURES I
LECTURE 5: SHORTEST PATHS IN VALUED GRAPHS

JAN HUBIČKA

1. Labeled graphs

We already know that the breath first algorithm can be used to find a shortes path between given vertices in (directed) graphs. If one consider a bit more real world example, where vertices of graph represent crossings in a city and edges are streets connecting them it is easy to see that often not all edges are equal. For this reason we will now consider graphs with edges of different lengths.

More precisely we equip a given graph $G = (V, E)$ $G = (V, E)$ with function $\ell : E \to \mathbb{R}$ defining the *length* (or *label*) of a given edge. This way we create an *edge valued graph* (sometimes also called *network*).

Given a walk W in graph G, the *length* of W, written as $\ell(e)$ is defined as

$$\sum_{e \in E(W)} \ell(e).$$

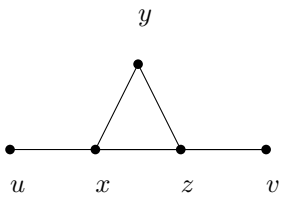
Given two vertices $u, v \in V(G)$ their *distance*, denoted by $d(u, v)$, is the minimum over lengths of all possible paths from u to v. Every path from u to v of length $d(u, v)$ is called a *shortest path* from u to v.

This is similar to definitions we introduced while analyzing BFS. There we proved that the length of shortest path (in unlabeled graph) is equal to the length of shortest walk. Repeating the same proof we get:

Lemma 1.1 (About simplifying walks)**.** *Let G be an labeled graph such that all lengths are non-negative. Then for every walk W from u to v there exists a path P from u to v such that $\ell(W) \leq \ell(P)$.*

From this multiple nice properties follows. Most importantly the "path distances" defined above are equivalent to "walk distances" defined by $d_w(u, v)$ behing the length of a *shortest walk* form u to v. We also get the triangle inequality: for vertices u, v, w it always holds that $d(u, v) \leq d(u, w) + d(w, v)$.

Example. Observe that non-negativity of edges is important here and the lemma is not true for a graphs containing edges of negative length in general. This is shown on the following example:



With all edges having length -1 the shortest path from u to v is u, x, y, z, v. But there is a shorter walk, for example u, x, y, z, x, y, z, v.

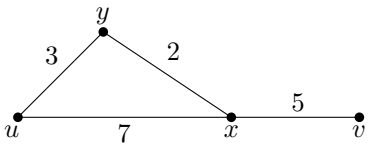
This does cause problems we will deal with in next class. In particular $d_w(u, v) = \infty$ while $d(u, v) = -4$. Triangle inequality also does not hold: $d(u, y) = -3$ and $d(y, v) = -3$.

Today we discuss an algorithm that works on graphs without negative edges. Next class we will see what can be done about negative edges.

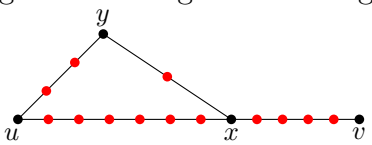
2. Dijkstra's algorithm

The most commonly used algorithm for finding shortest paths in valued graphs is the *Dijkstra's algorithm* (as you may have guessed it was introduced Edsger W. Dijkstra in 1956). We will try to jointly re-discover it starting from the following experiment.

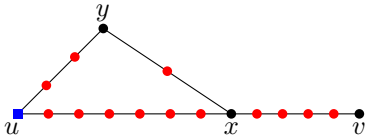
Example. Consider the following valued graph (numbers next to edges are their lengths):



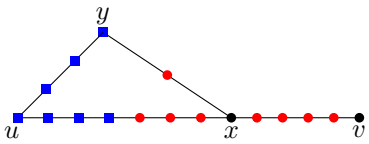
To determine the shortest path from u to v we could turn the valued graph into a normal graph and use BFS. This can be done by subdividing edges according to their lengths obtaining the following graph:



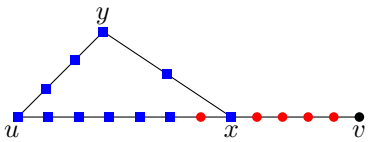
Red vertices are new vertices introduced to get distance between the original vertices right. Lets see what BFS from u will do. It will start in u (I will denote vertices visited by BFS using blue squares):



Then it will continue walking the graph until reaching vertex y. This happens after 3 steps (or iterations of the outer loop of the algorithm):

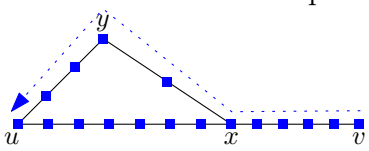


At steps 5 BFS will reach x.



And finally arrives to v at step 10 at which BFS terminates.

We thus determined that $d(u, v) = 10$ and moreover we can use the predecessor pointers to recover the shortest path.



The shortest path consists of the original vertices: u, y, x, v.

This algorithm has two main limitations.

(1) It works only with integer lengths and
(2) the running time depends on the sum of all lengths of all edges in the graph.

(Can you work out the actual running time?)

Dijkstra's algorithm avoids both problems by performing "discrete simulation" of this algorithm. Instead of going through all steps it only process those steps where something interesting is happening. By "interesting step" we mean all steps of the algorithm which reaches a new vertex of the original valued graph. My example above shows precisely all such interesting steps.

How this can be implemented? Given graph $G = (V, E)$ we will maintain, for every vertex $v \in V$, an "alarm" $h(v)$ which specifies time when algorithm will reach to this vertex. Because these times are not known at the beginning of the algorithm we will maintain $h(v)$ to be an upper estimate based on the part of graph we already visited. The Dijkstra's algorithm is implemented as follows:

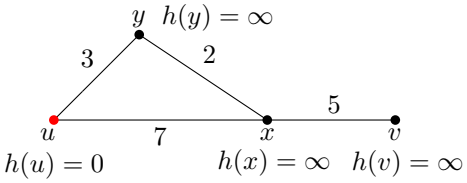
Algorithm (Dijkstra)**. Input:** Graph $G = (V, E)$, labeling of edges ℓ by non-negative reals and initial vertex v_0

1. For every vertex $v \in V$:
2. state$(v) \leftarrow$ unvisited
3. $h(v) \leftarrow \infty$
4. $P(v) \leftarrow$ undefined
5. state$(v_0) \leftarrow$ open
6. $h(v_0) \leftarrow 0$
7. Until there are open vertices:
8. Choose open vertex v with minimal $h(v)$

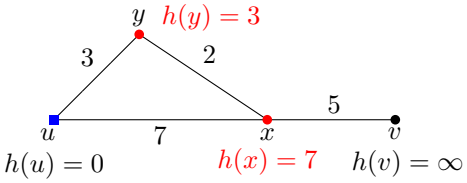
9. For every w such that $(v, w) \in E$
10. If $h(w) > h(v) + \ell(v, w)$ then:
11. $h(w) \leftarrow h(v) + \ell(v, w)$
12. state$(w) \leftarrow$ open
13. $P(w) \leftarrow v$
14. state$(v) \leftarrow$ *closed*

Output: Array of distances h, and array of predecessors P

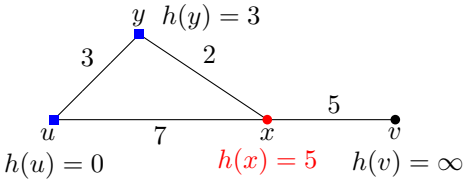
Example. On our sample graph starting from vertex $v_0 = v$ the algorithm will initialize as follows:



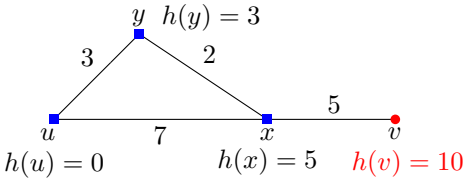
That is vertex v will be set as open and $h(v) = 0$. All other vertices are unvisited and their alarms are set to ∞.

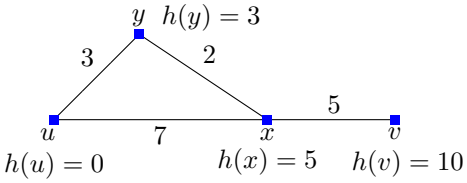


In next step we close vertex v and discover vertices x and y setting their alarms to length of edges from v.



Third step is interesting because algorithm discovers a shorter path to vertex x and re-visits its alarm.





In next two steps algorithm finishes discovering the shortest path of length 10.

2.1. **Correctness of the algorithm.** The correctness of the algorithm follows from discussion above: it computes what BFS would do

on the subdivided graph. Because Dijsktra's algorithm also works with non-integer lengths one can prove its correctness by showing the following three invariants which holds at the start of the outer loop (line 8):

(1) For every closed vertex v it holds that $h(v) = d(v_0, v)$
(2) For every open vertex v it holds that $h(v)$ is the length of shortest walk from v_0 to v with the property that all internal vertices (that is, all vertices of walk except for the first and the last one) are closed.
(3) Every vertex v reachable by a walk with all internal vertices closed is either open or closed.

I leave verification of these invariants as an exercise. It is very similar to the analysis of BFS we did before. Remember that you need to use the fact that all lengths are non-negative.

From these invariants and similar analysis as we did for BFS it follows that the algorithm will close precisely those vertices reachable from v_0 and set $h(v)$ to the length of shortest walk (or, equivalently, path) as intended.

2.2. **Time complexity.** Lets analyze time needed to execute individual parts of the program:

(1) Lines $1, 2, 3, 4$ (the initialization loop) executes in time $O(n)$.
(2) Lines $5, 6$ executes in time $O(1)$.
(3) Outer loop at lines $7, 8 \ldots, 14$ executes once per each vertex (because every vertex can be closed just once).
(4) Inner loop at lines $10, 11, \ldots 13$ overall executes at most $O(m)$ times.

This is all similar to the DFS and BFS algorithms. However there is an important problem with line 8. Simple implementation of line 8 will need to walk all vertices (to find minimal value of $h(v)$) and will run in $O(n)$. Line 8 is executed n times and thus needs $O(n^2)$ time overall. This is more than the rest of algorithm (that is $O(n + m)$) because $m \leq \binom{n}{2} \leq n^2$. From this we get:

Observation 2.1. *Dijkstra's algorithm will finish in time $O(n^2)$*

Fortunately this time can be improved. For this we will consider implementation of the alarms using an *data-structure* which will hold open vertices with their alarms (or keys).

Towards that we look into what Dijkstra's algorithm does with alarm: We perform the following operation on alarms:

(1) INITIALIZE we set alarm of a given vertex to specific value in line 3.
(2) DECREASE in line 11 we decrease value of a given alarm.

(3) ExtractMin in line 8 we look for minimal value across open vertices. We close the vertex later and thus we can also think of it as extracting from heap.

We will think of the heap as a "black box" that implements those operations with known time complexity. This lets us to formulate our time analysis more precisely and generally:

Theorem 2.2. *Dijkstra's algorithm will finish in time* $O(nT_i + nT_x + mT_d)$ *where* T_i, T_x, T_d *respecitvely are the time complexities of* Initialize, ExtractMin, *and* Decrease *respectively.*

A *heap* is data-structures which implements operations Initialize and ExtractMin. Operation Decrease is less standard and the datastructure is then usually also called an *heap* or *heap with decrease*. A simple implementation of heap is done by an array holding the *keys* (values $h(v)$ in our algorithm) where Initialize and Decrease are simple assignments (thus running in $O(1)$) and ExtractMin is a loop looking for the minimal element running in $O(n)$. By Theorem 2.2 we thus know that the algorithm with such implementation of heap will run in $O(n+n^2+m) = O(n^2)$ as we already shown in Observation 2.1.

From the algorithmization class you should know the binomial heap. If not, please review it, for example, at

https://en.wikipedia.org/wiki/Binomial_heap

Binomial heap implements Initialize and ExtractMin in time $\log(n)$. This heap can be extended to also implement Decrease in time $\log(n)$: for this you only need to observe that decreasing of a key of given element in heap can be done in the same way as Initialize. However it is necessary to solve the problem how to quickly find a given vertex v in the heap. For that we need to add a new array which, for given vertex v points to its current position in heap.

Corollary 2.3. *Dijkstra's algorithm implemented with binomial heap runs in time* $O((n+m)\log n)$.

This time is good for many real-world applications, but not optimal. For dense graphs we get $O(n^2 \log n)$ which is even slower than the runtime of the naive algorithm. Thus for sparse graphs (with few edges) biniomal heap wins, however for dense graphs (with many edges) the array wins. A smart solution to this problem is known as *d-ary* heap. See for example:

https://en.wikipedia.org/wiki/D-ary_heap

Even faster implementation of heap is the Fibonacci heap whch implements Initialize and ExtractMin with time $\log(n)$ but decrease is $\log(n)$. This datastructure is not simple and you will learn it in later semmester. If you are interested, see, for example

https://en.wikipedia.org/wiki/Fibonacci_heap

3. References

There are plenty of resources on Dijsktra's algorithm. I give few links:

(1) Wikipedia webpage
(2) Recorded lecture from UC Davis
(3) Lecture notes from Stanford
(4) Průvodce labyrintem algoritmů, sekce 6 (In Czech)
(5) Nahrávka Martinovy přednášky (In Czech)