Algorithms to compute shortest paths in graphs

Katarina Gačnik Mateja Smrekar

20. februar 2017

0.1 Description of the problem

Definition

- Dense graph is a graph $G = (V, E)$ in which the number of edges is close to the maximal number of edges i.e. $|E| = O(|V|^2)$.
- Graph $G = (V, E)$ with only a few edges is a sparse graph i.e. $|E| = O(|V|)$. It's number of edges is close to the minimal.

Our problem We will implement two of the standard algorithms (Dijkstra and Bellman-Ford) to compute shortest paths in weighted directed or undirected graphs and analyze and compare their running time on a sparse and dense graphs.

0.2 Dijkstra's Algorithm

Dijkstra's algorithm (discovered by E. W. Dijkstra) solves the problem of finding the shortest path from one verticle in a graph (we call it a source) to any other verticle. Hence we call this problem a single-source shortest paths problem. Algorithm works for weighted directed or undirected graphs that can containe cycles, but it doesn't work if any of it's edge has a negative weight.

Implementation of the algorithm in Sage that we wrote with added comments is written below:

```
def minimum(p, A):
    v = None
    razdalja = float('inf')
    for u in A:
        if p[u] <= razdalja: #1
            v = u
            razdalja = p[u]
    return v


def Dijkstra(G,s): #s is our starting vertex
    inf = float('inf') #we define the number infinity
    d = {u:inf for u in G} #2
    Pi = {u:None for u in G} #3
    d[s] = 0 #distance from starting vertex to itself is 0
    Q = set(G.vertices()) #4
    while len(Q)> 0:
        u = minimum(d, Q) #5
        Q.remove(u) #we remove u from the set Q
        for _, v, t in G.edges_incident(u): #6
            if v in Q and d[v] > d[u] + t: #7
                d[v] = d[u] + t
                Pi[v] = u
    return d, Pi
```

Comments:
#1 - If we use only <, it's possible that function returns $v = None$. That creates a problem later on in function Dijkstra.
#2 - We set infinity lenght for every vertex in graph G.
#3 - We set 'parents' or predecessor of every vertex to be $None$.
#4 - Let the Q be the set of every vertex in graph.
#5 - Function *minimum* finds the vertex in set Q which has the lowest value $d[]$.
#6 - G.edges_incident() returns all edges from vertex u. The first element of trinity is always u, so we don't need to remember it.
#7 - We limit ourselves to verteces v that are still in the set Q, so we get directed edges that interest us and make the relaxation step.

0.2.1 Running time

Running time of Dijkstra's algorithm depends on the way we store data in the set Q. We have several options but let us introduce the main ones.

Q as Linear Array

Function *minimum* finds the vertex in set Q with the smallest value $d[]$ in $O(V)$ time and there are $|V|$ such operations. Therefore, a total time in the *while* loop is $O(V^2)$.
Since total number of edges in all the adjancency list is $|E|$, therefore $for-$ loop iterates $|E|$ times with each iteration taking $O(1)$ time.
Hence, the running time for the algorithm with array implementation is $O(V^2 + E) = O(V^2)$.

Q as Binary Heap

Running time is in this case dominated by heap operations that are each taking $O(\log n)$. The heap operations that we do in Dijkstra's algorithm are updating and also deleting the minimum vertex and relaxation step.
We first note that building the priority queue takes $O(V)$ since we initially add every vertex in the graph in the set Q.
The *while* loop is executed once for every vertex and within that loop each call to *Q.delete* takes $O(\log V)$, therefore together it takes $O(V \cdot \log V)$.
The for loop is executed once for each edge in the graph, within it the call to relax values $d[]$ takes $O(\log V)$, hence together $O(E \cdot \log V)$.
Running time of the algorithm with binary heap is $O((V + E) \cdot \log V)$.

Q as a Fibonacci Heap

The main difference in this case is that operation in relaxation step now only takes $O(1)$ amortized time for each of the $|E|$ edges.
Together algorithem demands $O(E + V \cdot \log V)$ time.

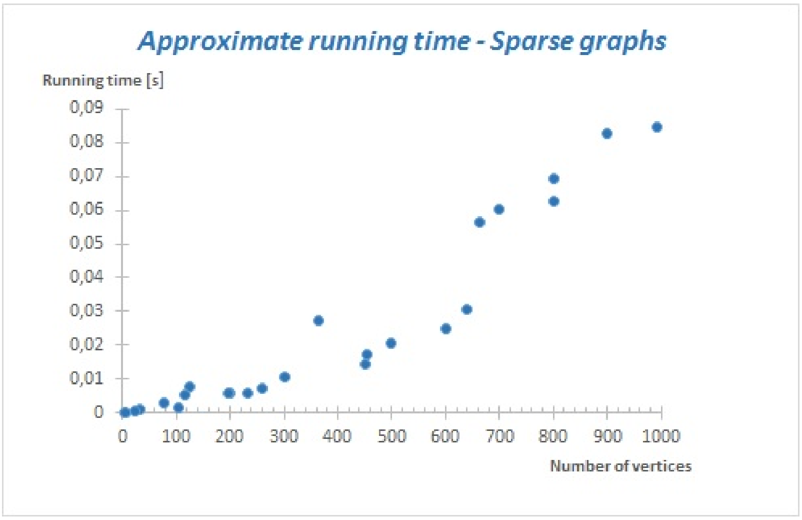
Analysis and Conclusion

The best possible running time for dense graphs we get by using Fibonacci Heap - $O(E+V \cdot \log V)$, but because in dense graphs applies $|E| = O(|V|^2)$, that gives

us runtime of $O(V^2 + V \cdot \log V) = O(V^2)$. A better runtime would be suprising since we have to look at every edge at least once.
On the other hand we can already get the best running time for sparse graphs if we simply use Binary heap - it gives us result in $O((V + E) \cdot \log V)$ time. Note that this time becomes $O(E \log V)$ if all verticles in graph are reachable from the source verticle.

0.2.2 Experiments

For the experimental part of this assignment we took various graphs from collections on the internet (with different number of vertex and density of edges), run them through algorithm and examined and compared running times. The graphs below show our results.



0.3 Bellman-Ford Algorithm

Bellman Ford algorithm calculates shortest path in weighted directed graphs from one source to all other vertices. Though it is slower than Dijkstra algorithm, it works on graphs that have negative edge weights, but fails if graphs have negative cycles. It was named after Richard Bellman and Lester Ford who published it in 1958.
The algorithm is based on the relaxation operation. Relaxation is repeated $|V| - 1$ times at most. First you approxiamate your solution, then step by step you replace it with more accurate values, until you reach the optimum solution. Two nodes are taken as arguments and weighted edge that connects them.

Input: directed, weighted(edge weights $wt_e : e \in E$ with no negative cycles) graph $G(V, E)$, source vertex $s \in V$.
Output: for all vertices u reachable from s, dist(u) and predecessor of u.

```
def bellman_ford(G, s):
    dist = {} #we create an empty dictionary of distances
    predecessor = {} #we  create an empty dictionary od predecessors
    V = G.vertices() #we create list of nodes
    E = G.edges()  #we create list of edges
    for v in V:    #1
        if v == s:
            dist[v] = 0
        else:
            dist[v] = Infinity
        predecessor[v] = 0
    for i in range(1, len(V)):    #2
        for (u,v,wt) in E: #2
            if dist[u] + wt < dist[v]:
                dist[v] = dist[u] + wt
                predecessor[v] = u
    for (u,v,wt) in E:     #3
        if dist[u] + wt < dist[v]:
               raise ValueError("Graph␣contains␣a␣negative−weight␣cycle")
    return dist,predecessor
```

Comments:
#1 - We create a loop that goes throgh all vertices of graph G. If vertex $v = s$ then $dist[v] = 0$, else $dist[v] = \infty$
#2 - this loop goes through each $e = (u, v, wt) \in E(G)$ and relaxes it. This is done |V| - 1 times.
Relaxation: If distance from the source vertex s to u plus wt (weight between nodes u and v) is less than distance from source vertex to v, then $dist[v] = dist[u] + wt$ and $predecessor[v] = u$.
#3 This loop checks for negative-weight cycles.

0.3.1 Running time

Running time depends on the number of relaxations calls.

step 1:

```
    for v in V:   #O(|V|)
        if v == s:
            dist[v] = 0
        else:
            dist[v] = infinity
        predecessor[v] = 0
```

Step 1 takes O(V) time.

step2

```
  for i in range(1, len(V)): #3
        for u,v,wt in E: #2]
            if dist[u] + wt < dist[v]:
                dist[v] = dist[u] + wt #1
                predecessor[v] = u #1
```

$\#1 = O(1)$
$\#2 =$ the code takes $O(|E|)$ time, where E is the number of edges.
$\#3 =$ relaxation is repeated $O(|V|) - 1$ times , where V is the number of vertices.

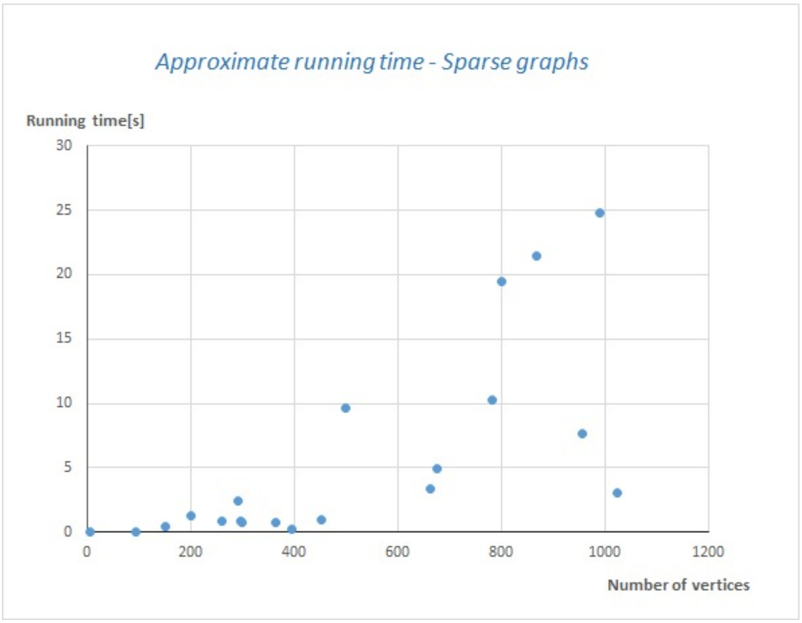
We can deduce from what wee see above, that step 2 of the Bellman-Ford algorithm makes E relaxations for every iteration, and there are $|V| - 1$ iterations, so in worst case scenario algorithm runs $O(|V||E|)$ time. Usually running time is much lower though. Best case scenario running time would be $O(|E|)$, in a graph where only one iteration is needed.

step3

```
for u,v,wt in E:
        if dist[u] + wt < dist[v]:
            raise ValueError("Graph␣contains␣a␣negative−weight␣cycle")
```

Step 3 takes $O(|E|)$ time.
Altogether running time is dominated by time $O(|V||E|)$.

0.3.2 Experiments



We can observe that running time is growing with the number of vertices, but also very much depends on the number of edges. So running time in a graph that has around 500 vertices can be a lot bigger then running time in a graph that has 1000 vertices, if the number of edges in the first graph is much bigger than in the second one.