Longest path in a DAG

Assignment of Graph Theory and Algorithms

Alessandro Bregoli

Notation

A **directed graph** $G = (V, E)$ is composed by a finite set of **nodes** V and a finite set of **edges** E such that $E \subseteq [V]^2$. A graph is weighted if there is a function $w : E \to \mathbb{R}$. Given two nodes $x, y \in V$ we say that x is **adjacent** to y if $\{x, y\} \in E$. This relation isn't symmetric. A **path** is a sequence of adjacent vertices $\langle v_1, v_2, ..., v_k \rangle : v_i \in V \wedge \{v_i, v_{i+1}\} \in E$. The **weight** of a path is equal to the number of edges in the case of a non-weighted graph while it is equal to the sum of the weights of the arcs in the case of a weighted graph. A path is **simple** if all vertices in the path are distinct. A path for a **cycle** if $v_0 = v_k$ and the path contains at least one edge. A Directed Acyclic Graph (**DAG**) is a directed graph without cycles of any length. A **shortest simple path** is a simple path of minimal weight. Conversely, a **longest simple path** is a simple path of maximum weight.

Single source longest simple path problem

The single source shortest path problem can be solved in polynomial time for each graph. On the other hand, the single source longest simple path is NP-hard for a generic graph. However, if we only want to solve the problem for directed acyclic graphs we can modify the algorithm for finding the single source shortest path in a convenient way and identify the single source simple longest path in linear time.

The algorithm

Both the single source shortest path and the single source longest path for a directed acyclic graph are based on two components: a relax procedure and a topological sort algorithm. The only difference between the two algorithms is the definition of the relax function.

- **Relax procedure for the single source shortest path algorithm**. The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found, so that far by going through u and, if so, updating the shortest path.

- **Relax procedure for the single source longest simple path algorithm**. The process of relaxing an edge (u, v) consists of testing whether we can improve the longest path to v found, so that far by going through u and, if so, updating the longest path. (Algorithm 1)

Algorithm 1 RELAX function

```
function RELAX(v, w ∈ G.V, w: edge weight)
    if v.longest_path < u.longest_path + w(u,v) then
        v.longest_path = u.longest_path + w(u,v)
        v.predecessor = u
    end if
end function
```

The **topological sort algorithm** is an algorithm witch find an order among the nodes such that, if there is and edge (u, v) then u appear before v in the topological ordering. As shown in 2) , the topological ordering is based on the DFS-visit and inherits its computational complexity.

Algorithm 2 TOPOLOGICAL SORT

```
function TOPOLOGICAL_SORT(G)
    topological_order = LIST()
    for each u ∈ G.V do
        u.color = WHITE
        u.predecessor = NIL
    end for
    for each u ∈ G.V do
        if u.color == WHITE then
            DFS-VISIT(G, u, topological_order)
        end if
    end for
    return topological_order
end function

function DFS-VISIT(G, u, topological_order)
    u.color = GRAY
    for each v ∈ G.Adj[u] do
        if v.color == WHITE then
            v.predecessor = u DFS-VISIT(G, v, topological_order)
        end if
    end for
    u.color = BLACK
    topological_order.prepend(u)
end function
```

Algorithm 3 Single source shortest path

```
1: function DAG-LONGEST-PATHS(G, s)
2:     T = TOPOLOGICAL_SORT(G)
3:     for v ∈ G \ s do
4:         v.longest_path = −∞
5:     end for
6:     for v ∈ G do
7:         v.predecessor = NILL
8:     end for
9:     s.longest_path = 0
10:    for u ∈ T do
11:        for v ∈ G.Adj[u] do
12:            RELAX(u,v,w)
13:        end for
14:    end for
15: end function
```

By exploiting the topological ordering it is possible to define the **single source longest path algorithm** as shown in Algorithm 3. The rationale behind this algorithm is based on the fact that there are no cycles in a DAG. Consequently if the dag contains a path from vertex u to vertex v, then u precedes v in the topological sort. For this reason it is sufficient to make just one pass over the vertices in the topologically sorted order.

Time Complexity

The goal of this subsection is to show the linear complexity of the single source longest path algorithm. In order to achieve this goal we can split the algorithm in its main components described in Algorithm 1, Algorithm 2 and Algorithm 3.

Analyzing the RELAX function in Algorithm 1 we can say that the time complexity is: $O(1)$.[1]

It is well known from the literature that a DFS search has a time complexity of $O(V + E)$. Since the TOPOLOGICAL SORT presented in Algorithm 2 is basically a DFS also its time complexity is $O(V + E)$.

In order to find the time complexity of the Single source shortest path presented in Algorithm 3 we need to split the code as follow:

- **Line 2**: it calls the TOPOLOGICAL SORT algorithm with time complexity $O(V + E)$
- **Lines 3 - 9**: these lines initialize the data structures for each node with a time complexity of $O(V)$.

[1] This is true only if the access to the weight $w(u, v)$ is $O(1)$. For this reason the selection of an appropriate data structure is crucial.

- **Lines 10 - 14**: in this portion of code the algorithm pass through each node and each vertex exactly once. For this reason the time complexity is $O(V+E)$

Combining the complexity of each piece of code we have that the complexity of the algorithm is:

$$O(V+E)$$

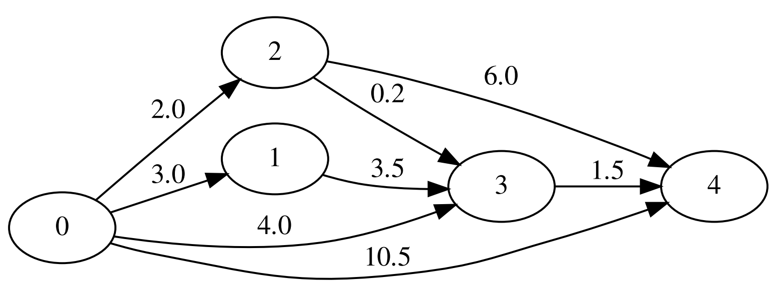
Implementation

The previous section shows that the loongest single path algorithm has a complexity of $V(V+)$. However this is true only if we use the correct data structure to represent the graph. For example if we use an adjacency matrix both the Algorithm 2 and the nested for of the Algorithm 3 become $O(V^2)$. For this reason I decided to use a adjacency list implemented with a linked list.

The programming language that I decided to use is Rust[2] mainly becouse of its high performance. The implemented algorithm is available on github[3] and is capable of:

- Load the structure of a network from an edge list file where each line is structured as follow: Source, Destination, weight.
- Return a topological order of the nodes for the loaded network.
- Compute the single source shortest/logest path given a network and a source node.

Demonstration



In Figure there is the dag that we will use in this section. First of all we need a file containing the edge list representation of the network: Listing 1.

[2] https://www.rust-lang.org
[3] https://github.com/AlessandroBregoli/rdag

Listing 1: Edge List

```
0	1	3.0
0	2	2.0
0	3	4.0
0	4	10.5
1	3	3.5
2	3	0.2
2	4	6.0
3	4	1.5
```