Dijkstra's algorithm: an interactive web application

Riccardo Locci

July 2020

Abstract

Dijkstra's algorithm is an algorithm useful to find the shortest paths between nodes within a graph. It was conceived by Edsger Dijkstra in 1956 and published three years later.[1] Focus of this work is to propose the implementation of Dijkstra's algorithm as an interactive web application that shows the algorithm's execution flow step-by-step. This work will result in a auxiliary learning tool to better understand and visualize how the algorithm works thanks to the grafical user interface exposed by the application. Since the developed application is a client-only web application, it targets only problems with low cardinality in terms of number of nodes (working well with $N \leq 100$). The application was develop and tested on Google Chrome on the following machines:

- Desktop:
  - OS: Windows 10 Pro 1903,
  - CPU: Intel®Core i5-4440 3.10 GHz
  - RAM: 16 GB DDR3
- MacBook Air:
  - OS: macOS Catalina 10.15.5,
  - CPU: Intel®Core i5 dual-core 1.6 GHz
  - RAM: 8 GB LPDDR3

1 Shortest Path Problem formal definition

First, we will give Shortest Path Problem's formal definition through its mathematical model.

We're considering a directed graph $G = (N, A)$ where:

- N: is the set of nodes
- A: is the set of arcs
- n: is the number of nodes
- m: is the number of arcs
- a_{ij}: is the arc which links the node i and the node j
- c_{ij}: is the cost of the arc a_{ij}
- $d(i)$: is the distance from the node i
- $pred(i)$: is the predecessor of the node i

We assume that the graph G is a directed graph and the costs c_{ij} are only integer values. To solve a Shortest Path Problem we must find the shortest path between a node s (called *source*) and a node t (called *sink*).

The objective function is defined as:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

and it is subject to the following constraints:

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = \begin{cases} 1, & i = s \\ -1, & i = t \\ 0, & otherwise \end{cases} \tag{1}$$

$$x_{ij} \geq 0, \forall (i,j) \in A$$

In a more generic version of the problem, we will have that all the nodes (except for s) will be sinks. We need to adapt the mathematical model to the new problem, so (1) becomes:

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = \begin{cases} n-1, & i = s \\ -1, & \forall i \in N - \{s\} \end{cases}$$

2 Overview about Dijkstra's Algorithm

As previously said, Dijkstra's algorithm is an algorithm to find shortest paths between nodes within a graph and it belongs to the class of label setting algorithms. In particular, we want to find the *shortest path* from a node s (called *source*) to a node t (called *sink*). At each iteration, nodes are divided into a set of *permanent nodes* and a set of *temporary nodes*: permanent nodes are nodes whose labels cannot be improved because they are already optimal, temporary nodes are nodes whose labels are not proved to be optimal[2]. The distance label to any permanent node represents the shortest distance from the source to that node. For any temporary node, the distance label is an upper bound on the shortest path distance to that node. The basic idea of the algorithm is to fan out from node s and permanently label nodes in the order of their distances from node s[2]. Initially, we give node s a permanent label of zero, and each other node j a temporary label equal to ∞. In label settings algorithms, at each iteration, a temporary node is selected so that it is removed from the temporary nodes set and inserted in the permanent nodes set[2].

2.1 Algorithm

The idea is to select a feasible temporary node i and move it in the permanent nodes set updating the distance labels of nodes linked with i. Specifically, what we do is to select a temporary node i with minimum distance label and make that node a permanent one, check its adjacency list $A(i)$ and update the distance labels of the nodes linked with i. When we update the distance labels, we use the equation:

$$d(j) = d(i) + c_{ij}$$

where $d(j)$ and $d(i)$ are respectively the distances of node j and node i from the source. If we update the distance label, then we update the precedent of node j as follows:

$$pred(j) = i$$

2.2 Pseudocode

The following pseudocode[2] represents the classical version of the algorithm, which will be used as a kickstart for the final web application:

Algorithm 1 Dijkstra's Algorithm

```
1:  procedure DIJKSTRA
2:      S := ∅, S̄ := N
3:      d(i) := ∞, ∀i ∈ N
4:      d(s) := 0, pred(s) := 0
5:      while |S| < n do
6:          let i ∈ S̄ be a node for which d(i) := min{d(j) : j ∈ S̄}
7:          S := S ∪ {i}
8:          S̄ := S̄ − {i}
9:          for (i, j) ∈ A(i) do
10:             if d(j) > d(i) + c_ij then
11:                 d(j) := d(i) + c_ij
12:                 pred(j) := i
```

3 Implementation of the algorithm

In this section we will discuss the proposed implementation, but first there are some aspects that require some clarifications. It is not possible to perform a completely random generation of a graph, since it must be a graph on which is possible to perform a Shortest Path Problem solving algorithm. Because of this, the application allows to generate a random graph valid as input for Dijkstra's algorithm, without the risk of infinite loops.
Another point of discussion will be the algorithm itself: to allow the application to behave in the right way, **Algorithm 1** will be split in multiple steps.
Finally, since the target of this work is to create a graphical user interface to illustrate and better understand Dijkstra's algorithm, computational complexity was not taken into account: this obviously leaves space for future optimizations.

3.1 Data structures

The application uses simple lists of nodes and arcs to represent the graph. This choice is partly forced by the library[3] used to draw the graph in the graphical user interface, since the library requires this exact data structure as input. Considering that the target of this work was to implement Dijkstra's algorithm, the role of the drawings was delegated to a ready-to-use library to cut development times.

A sample graph is then represented by the following JSON file:

```
{
    "nodes": [
        {
            "id": 1,
            "title": "1"
        },
        {
            "id": 2,
            "title": "2"
        }
    ],
    "edges": [
        {
            "source": 1,
            "target": 2,
            "cost": 5
        }
    ]
}
```

Listing 1: JSON file example

The result will be the following graph:

Figure 1 Graph drawn from Listing 1

This data structure is not efficient when implementing search functions for nodes or arcs. Thus, to improve code quality, the first operation after file generation will create some support data structures:

- A JavaScript object that works as an index to find nodes;

- A list for each node to instantly get its leaving star.

Moreover, thanks to these additional data structure we can retrieve useful information in constant time.

3.2 Custom implementation

Classical implementations of Dijkstra's algorithm work as a unique flow that from an input graph computes shortest paths and return them as their output. This work will obviously return the same output of a classical implementation, but it will also show every step that leads to the computation of every path. It follows that the algorithm cannot work as a single block of code, but it needs to be split and modularized in multiple phases.

The following sections will cover the content of the three phases of the custom algorithm. The algorithm works in two modes: single-target mode and multi-target mode; in multi-target mode the algorithm marks all the nodes (except for the source) as sinks and skips the third phase (see Section 3.2.3).

3.2.1 Preprocessing

Preprocessing is the first phase that will run after input is provided. Its role is to populate the supporting data structures and initialize other data structures used by other features (i.e.: navigable algorithm's history). It also covers the first steps of **Algorithm 1** (rows 2-4) and updates the graph in the canvas to highlight the source node and (in single taget mode only) the sink node, and to initialize distance labels in the canvas.

3.2.2 Processing

This phase is the longest and the most important one since it covers the remaining instructions on **Algorithm 1**. Being the most complex phase, it is split in a series of steps, and every step is composed by a set of substeps. A new step begins every time we select a node from the temporary nodes set and ends when we have visited each node directly linked to the one that started the current step. Every time a new step begins, the application updates the graph in the canvas by highlighting the colour of the current node. Accordingly, within a step, a substep begins every time we visit a new node from current's node leaving star: during each substep the application performs all the checks needed to update the (temporary) shortest paths and distances from the source and shows what changed by:

- updating colours and distance labels of every new reached node;
- updating the info box with the list of changes;
- updating the table of the (temporary) shortest paths

In short, every iteration of the outer loop (*while* loop at line 5 of Algorithm 1) is treated as a distinct step; moreover, each iteration of the inner loop (for loop at line 9 Algorithm 1) runs as a separate substep within the same step.

3.2.3 Postprocessing

This phase is run only in single-target mode to show how the shortest path from the source to the sink is found. Indeed, the application recursively fetches the precedent of each node in the shortest path starting from the sink until it gets to the source. Every time it jumps from a node to its precedent, the application highlights the nodes and arcs involved in the path.

We can say that the application partly applies postprocessing in multi-target mode: once we get to the end of the processing history, the application allows to highlight one path at a time by interacting with the shortest paths table.

3.3 Web application GUI

Within this section we will show how the application appears and which parts compose the Graphical User Iterface: we will talk about all the possible interactions and features, and how they influence the understanding of the algorithm.

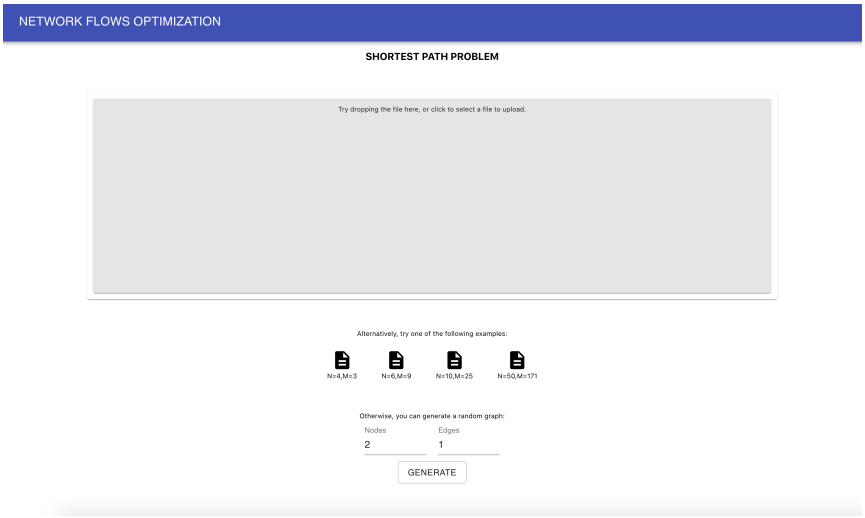

Figure 2 Landing page and input choice

3.3.1 Input choices

First page of the web application[4] allows multiple ways to provide a graph as input for the algorithm. The page is divided in three sections:

- a **drop area** where to drop a properly formatted JSON file describing a graph (see Listing 1): it is also possible to select a file from file explorer by clicking over the area;
- a small set of **sample graphs** on which the application was tested: every file presents a name that points the number of nodes and arcs;
- a **generation form** that allows the user to create a pseudo-random graph.

The generation form needs two inputs: the number of nodes and the number of arcs. The maximum number of nodes is limited to 50 since a greater number would be disadvantageous for multiple reasons:

- being the focus of this work strongly graphical, the drawn graph must be understandable: too many nodes or arcs will limit this aspect;
- to completely execute the algorithm the user would need to go through every step by manually clicking on the **Next** button (see Section 3.3.3): this would be frustrating when working with a very big graph;
- the chosen drawing library is not completely suited to excessively customize the available graph entities: to draw them as it was done, every entity needs to be rendered twice. So, even if the algorithm runs smoothly, when the number of entities is too big the GUI shows some rendering slowness.

Moreover, the number of nodes limits the number of arcs according to the following inequality:

$$m \leq \frac{n * (n-1)}{2}$$

where m is the number of arcs and n is the number of nodes.

The generator will then create a graph with the following pattern:

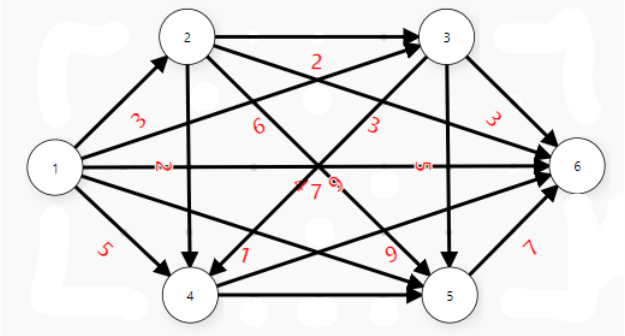

Figure 3 Random graph with n=6 and max m

that is, node i-th's leaving star only contains nodes with id $> i$.

This pattern ensures that graph's topology never contains loops, so that the algorithm can work without problems.

3.3.2 Canvas

After input is provided, the graph will be drawn in the canvas: its purpouse is to help understanding what is happening while the algorithm runs. Nodes are positioned in such a way that they would not interfere with the comprehensibility of the topology, however this is not always possible (e.g. nodes having too many entering or leaving arcs): to handle these situations, it is possible to manually move nodes. Each node is drawn as a circle containing its id in the center, and border

colour of the circle indicates the status of the node: each status is described in the legend. Moreover, below each node a number can be found which represents the distance label of that node from the source. Arcs are drawn as arrows that start from their source and end to their target: they can assume a subset of statuses compared to nodes but the colours refer to the same legend of nodes. Near the center of each arc there can be found its cost, if its value is greater than zero. On the left-below corner of the canvas is placed a minimal menu of two elements:

- a **sliding bar** to zoom the canvas in or out, useful when the number of nodes is too high to display them all;
- a **center button** that computes the center of gravity of the graph and accordingly repositions the canvas;

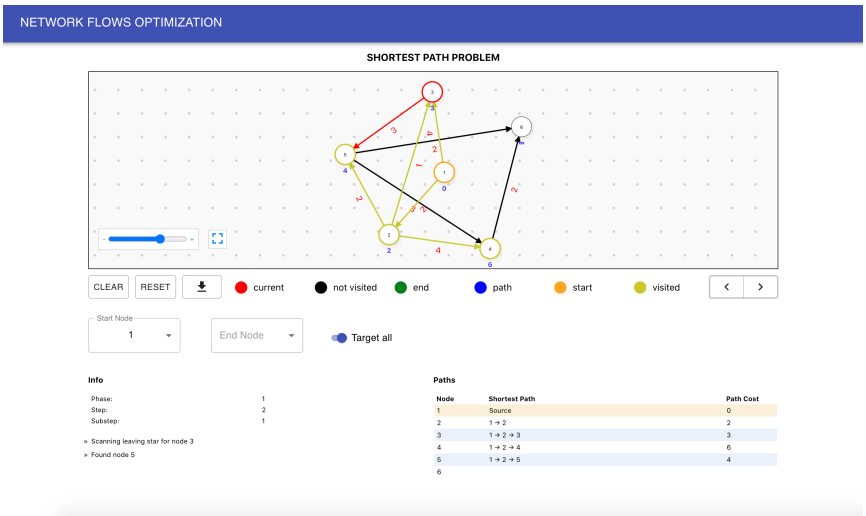

Figure 4 Application GUI example

3.3.3 Interaction section

Below the canvas there are some elements that help to iteract with the application:

- the **clear button** reverts the current state of the application and sends the user back to the input page;
- the **reset button** reverts the state the application to the point where input was just being provided;
- the **download button** allows to download the current drawn graph as a JSON file;
- the **legend** describes the colours in the canvas and allows to change them by clicking on each status;
- the **navigation buttons** are used to keep executing the algorithm and visualize its history.

- the **source/sink(s) selectors** are used to specify the source node and the target node(s).

Every time the algorithm has to keep executing, the user need to press on the **Next** button (the navigation button on the right): the new state is computed only once then it is stored within a proper data structure. By storing every state, the application allows to navigate algorithm execution history back and forth.

3.3.4 Info section

The last section provides written descriptions and tabular representations of what it is happening in a given state of the algorithm.

On the left, there is a **Info** box that shows in which state the algorithm is with the phase number, step number and substep number: these three numbers help the user referring to a specific point in time within an execution. The last part of this section is a written list of actions that were performed during that particular state: this information is stored with other ones regarding the state (such as status of nodes and arcs, temporary paths, distances).

On the right, the application shows a **Paths** table that lists the path to reach every node discovered until that point in time and their distance from the source. If the selected state is the last performed by the algorithm, it is also possible to click on one particular row to display that path on the canvas.

4 Results and performances

This application can successfully run a (properly customized) classical implementation of Dijkstra's algorithm for shortest path problems. It was tested on a set of graphs that satisfy the topology provided in Section 3.3.1 and on **IBM ILOG CPLEX Optimization Studio** [5]: to compare the results a Python script (see GitHub repository [6]) was developed to convert JSON files to CPLEX models and then the resulting objective functions were compared. All the tests returned the same results on both the web application and IBM ILOG CPLEX Optimization Studio.

The application does not allow to jump to the end of the execution since its objective is to illustrate how the algorithm behaves at every step, so its performaces were not taken into account (the principal data structure used were lists). However, it is true that at every step the graph needs to be drawn again because of the library[3] that was used; moreover the computational effort needed to draw the graph further increases because the library does not provide a conventional way to heavy customize graph's entities, so each one needs to be drawn again after first rendering. This is not perceivable on a simple graph, but the application shows its limits while working with too large inputs.

5 Conclusions

The developed web application described in this work allows to correctly run Dijkstra's classical algorithm on graphs provided as JSON files or generated through the application's generator, and to see a step-by-step execution thanks to a user-friendly interface that at each iteration draws the graph and describes what it is

really happening from the beginning to the end. Moreover it is possible not only to proceed forward into the execution, but to go back in time, too: this feature allows a better understanding and analysis of the algorithm and provide a different perspective on particular topologies. The possibility to select and highlight a particular shortest path allows to check multiple targets and analyze how they are reached when stating from a particular source.

At this point in development, the application is surely useful when studying Dijkstra's algorithm, but there are many optimizations and features that can be considered to improve this work. A custom drawing library developed to precisely address the needs of the application, a better organization of the interface to improve user experience, multiple algorithms to choose, support for other input formats (e.g.: direct support for CPLEX models): this are just some example of how the application can be extended in the future.

References

[1] Wikipedia, *Dijkstra's algorithm*, `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`

[2] R.K. Ahuja, T.L. Magnanti and J.B. Orlin *Network Flows theory, algorithms, and application*

[3] GitHub, *React Digraph*: `https://github.com/uber/react-digraph`

[4] Project Web Application: `https://riccardolocci.github.io/NFO`

[5] CPLEX, *IBM ILOG CPLEX Optimization Studio*, `https://www.ibm.com/it-it/products/ilog-cplex-optimization-studio`

[6] GitHub, *Project Repository*, `https://github.com/riccardolocci/NFO`