Report on

Implementation of a Min Cost Max Flow Algorithm Using Dijkstra's Algorithm

Jan-Niklas Topf (E-Mail: jan-niklas.topf@etu.unice.fr)

April 1, 2025

Contents

1 Abstract

This report presents a Python implementation of a flow network solver supporting both Maximum Flow and Minimum-Cost Maximum-Flow computations. The algorithms used include a capacity-aware Dijkstra approach (widest path) and the Successive Shortest Path algorithm with reduced costs and node potentials. Additionally, the solver provides minimum cut detection and automated residual graph visualization via Graphviz.

2 Algorithms Used in the Flow Network Implementation

The implementation of the `FlowNetwork` class supports two central algorithms in flow theory: the **Maximum Flow** algorithm using a capacity-aware Dijkstra (widest path) [1], and the **Minimum-Cost Maximum-Flow** algorithm via successive shortest augmenting paths (SSAP) [1]. Additionally, the framework allows for **minimum cut detection** after computing the flow, and provides visualization of residual networks through Graphviz DOT export [2]. Detailed documentation of the code can be found in the source file.

2.1 Maximum Flow via Widest Path

To compute the maximum flow between a designated source and sink, a modified version of Dijkstra's algorithm is employed [1]. Instead of minimizing distances, this algorithm maximizes the bottleneck capacity along paths, i.e., the minimum residual capacity among the edges of a path. In each iteration, the algorithm searches for such a widest path using a max-heap priority queue. Once such a path is found, the flow is augmented along it by the bottleneck capacity, and the residual capacities are updated accordingly.
This process repeats until no more augmenting paths with positive capacity exist in the residual graph.

2.2 Minimum-Cost Maximum-Flow via Successive Shortest Paths

The minimum-cost maximum-flow problem is addressed using the **Successive Shortest Augmenting Path (SSAP)** method [1]. To handle different cost edges between two nodes, the function to read the graphs from the file automatically adds an intermediate node to prevent multiple arcs from one node. In this approach, each edge has an associated cost per unit flow, and the objective is to maximize the flow from source to sink while minimizing the total cost. To ensure correctness even in the presence of negative edge costs, *reduced costs* are used. These are computed using a set of node *potentials*, defined as [1]:

$$c_r(u, v) = c(u, v) + d(u) - d(v)$$

where c_r is the reduced cost, and $d(u)$ and $d(v)$ are the current potentials of nodes u and v, respectively.
In each iteration, Dijkstra's algorithm is used with these reduced costs to find a shortest augmenting path in terms of cost. If a path is found, the flow is augmented by the path's bottleneck capacity, and the total cost is increased

accordingly. The node potentials are then updated to preserve non-negative reduced costs for future iterations. This process continues until no more augmenting paths exist.

2.3 Minimum Cut Detection

After computing a maximum flow, the corresponding minimum cut can be extracted from the residual graph. A breadth-first search (BFS) is performed from the source node, following only those edges with positive residual capacity. The set of reachable nodes forms one side of the cut. All edges in the original graph that go from a reachable node to a non-reachable node constitute the minimum cut set. This technique relies on the **Max-Flow Min-Cut Theorem**, which states that the value of a maximum flow equals the capacity of the minimum cut.

2.4 Graph Visualization

To aid in debugging and analysis, the implementation supports automatic visualization of the residual network at any stage of the algorithm. This is achieved via the DOT language and the Graphviz toolkit [2]. The residual graph is exported as a `.dot` file and converted into a `.pdf` using the `dot` command-line tool. Forward residual edges are represented as solid black arrows, while reverse flows (from residual edges) are shown as dashed blue arrows. Edges are labeled with residual capacities and either original or reduced costs, depending on the context. Furthermore, nodes can be color-coded to reflect membership in a minimum cut or to distinguish auxiliary nodes introduced during graph transformation.

3 Execution of the Flow Network Solver

The implemented Flow Network Solver is written in Python and can be executed from the command line. The input file must contain the network structure in a predefined format.

3.1 Running from the Command Line

The solver can be launched with any input file using the following command:

```
python Project_JanNiklasTopf_2025_FlowSolver.py path/to/input_file.txt
```

If no file path is provided as an argument, the program will prompt the user to enter one interactively.

3.2 Input File Format

The input file must follow this structure:

- First line: `<number_of_nodes> <number_of_edges> <source> <sink>`
- Each subsequent line: `<start_node> <end_node> <capacity> <cost>`

Example [1]:

```
7 10 0 6
0 1 16 58
0 3 13 40
1 2 5 46
1 4 10 45
2 3 5 58
2 4 8 33
3 2 10 60
3 5 15 46
4 6 25 72
5 6 6 68
```

3.3 Output

After execution, the solver computes and prints several key results. First, it calculates the maximum flow from the source to the sink using a capacity-aware version of Dijkstra's algorithm, also known as the widest path algorithm. Next, it determines the minimum cut of the network based on the final residual graph. Finally, it computes the minimum-cost maximum flow using the Successive Shortest Path algorithm, which relies on reduced costs and node potentials to find augmenting paths with minimal cost. These results are printed to the terminal, and if Graphviz is installed, visual representations of the residual graph before and after the computations are also generated as PDF files.
In addition, visualizations of the residual graph before and after the computation are generated as PDF files (requires `Graphviz` [2] to be installed). The visualization of the iterations can be switched on or off in the main method of the code to enhance the understanding of the functionality of the code and the used algorithms.

References

[1] JC Régin. *Lecture: Operations Research*. 2025.

[2] Graphviz Development Team. *Graphviz - Graph Visualization Software*. 2025. URL: https://graphviz.org (visited on 04/01/2025).