Scientific Computing Project

Comparison of Single-Source-Shortest-Paths' algorithms and their implementation

Group 13 - Thomas Bake

Abstract

This project aims to compare and analyze different algorithms that solve the single-source-shortest-paths problem. A small project was developed in C++ to juxtapose the algorithms and measure their actual computation time. The results showed that there is a tendency for the Dijkstra's heap implementation and the Shortest Path Faster Algorithm (SPFA) to perform better than the naive variant of Dijkstra and Bellman-Ford, and this trend becomes more pronounced as the graph grows in size. The SPFA displays a good performance and its computation time is on the same order of magnitude as that of Dijkstra's heap variant. The findings suggest that the asymptotic runtime of an algorithm does not always reflect its actual computation time and a deeper analysis of the problem setting and other factors is required.

1 Introduction

The purpose of this project is to analyze and implement different algorithms that solve the single-source-shortest-paths problem on a common setting to study to what extent do they differ from each other and assess, how well does the asymptotic runtime translates into actual computation time in practice.

In order to study the previously mentioned points, a small project was developed in C++, which among others, contains implementations of a graph, the algorithms to be compared and an experiment setting for users to run tests in diverse manners, as well as for computing shortest paths and their corresponding costs employing the algorithm of their choice.

2 Methods

2.1 Theoretical background

In the following, a general overview of the problem theoretical setting of the problem will be presented, in order to provide a consistent and accurate analysis of the individual algorithms, which will provide a basis for the subsequent sections. However, no proofs of the mathematical statements will be provided, as there is extensive literature on the topics and it is not the main purpose of this study.

2.1.1 Single source shortest path problem

In the general shortest path problem a directed graph (digraph) $\mathcal{D} = (\mathcal{V}, \mathcal{A})$, a node $s \in \mathcal{V}$, and arc costs $c_a \in \mathbb{R}^{\mathcal{A}}$ are given, where $\mathcal{V}$ and $\mathcal{A}$ are the set of nodes and arcs, respectively. The task then, consists on finding the shortest s-v-paths for all $v \in \mathcal{V}$. A natural question to ask in this setting is whether or not such a dipath exists, but it is known from graph theory, that such a path (if there is one) can be computed by using breadth-first-search in linear ($\mathcal{O}(\mathcal{V} + \mathcal{A})$) time.

The basic idea underlying algorithms that solve the shortest path problem is based on the following observation:

Observation 2.1: If y_v is the least cost of a diwalk from s to v, then

$$y_v + c_{(v,w)} \geq y_w \quad \forall (v, w) \in \mathcal{A},$$

where $y \in \mathbb{R}^{\mathcal{V}}$ is called a potential.

Therefore, it is natural to define the following concept:

Definition 2.2: A vector $y \in R^{\mathcal{V}}$ is called a feasible potential if

$$y_v + c_{(v,w)} \geq y_w \quad \forall (v, w) \in \mathcal{A},$$

From this definition, the next lemma and corollary provide the motivation for the aforementioned observation, which is then exploited by all of the considered algorithms:

Lemma 2.3: If y is a feasible potential with $y_s = 0$ and P an s-v-walk, then $y_v \leq c(P)$, where $c(P)$ is the sum of the cost of each arc in P.

Corollary 2.4: If y is a feasible potential with $y_s = 0$ and P an s-v-walk of cost y_v, then P is a least-cost s-v-walk.
In other words, feasible potentials provide lower bounds for least cost paths. The next lemma states that subpaths of shortest paths are least cost paths themselves.

Lemma 2.5: Let $Pv_1, v_k = (v_1, v_2, ..., v_k)$ be a shortest path from vertex v_1 to vertex v_k and let let Pv_i, v_j denote the subpath of Pv_1, v_k from vertex v_i, to vertex v_j for any $1 \leq i \leq j \leq k$. Then, P_{v_i,v_j} is a shortest path from v_i to v_j.

2.1.2 Common subroutines

In the following, the four compared algorithms are presented with their respective asymptotic running time, as well as the similarities and differences between them.

First of all, it is worth highlighting that all four algorithms share the same initialization and a subroutine, which will be denoted as INITIALIZE-SINGLE-SOURCE and RELAX, respectively. For each algorithm one considers two sets or arrays associated to each of the nodes in the graph, namely those responsible for storing the predecessors ($\pi[v]$) and distances (or potentials) ($y[v]$). In the following listing, the pseudocode of both routines is layed out:

Algorithm 1 INITIZALIZE-SINGLE-SOURCE($\mathcal{D} = (\mathcal{V}, \mathcal{A})$, s)

```
1: for each vertex $v \in \mathcal{V}$ do
2:     $y[v] \leftarrow \infty$
3:     $\pi[v] \leftarrow$ NIL
4: end for
5: $y[s] \leftarrow 0$
```

Algorithm 2 RELAX($v, w, c_{(v,w)}$)

```
1: if $y[v] > y[v] + c_{(v,w)}$ then
2:     $y[w] \leftarrow y[v] + c_{(v,w)}$
3:     $\pi[w] \leftarrow v$
4: end if
```

First, it is worth highlighting that our vector of potentials (i.e. our set of distances) maintains an upper bound on the cost a shortest path from source s to v. However, lemma 2.3 states that, if such a potential is feasible, it serves as a lower bound for the cost for the path from s to v.

The relaxation process described by Algorithm 2 focuses on decreasing the shortest path distance between nodes and updates the predecessors' field accordingly. It is easy to observe that this process creates a feasible potential as described in Definition 2.2.

Each of the considered algorithms call these routines, but differ (among other aspects) on the amount of times that the relaxation is performed. Now we have the necessary ingredients to present each SSSP-algorithm that was considered in this project.

2.1.3 Algorithms

In the following, let $n = |\mathcal{V}|$ and $m = |\mathcal{A}|$. Then, the table below summarizes the settings in which each algorithm is used as well as their asymptotic running time.

Perhaps the least-known algorithm considered here is Shortest Path Faster Algorithm (SPFA), originally published by Edward F. Moore as "Algorithm D" in 1957, which is an improvement of Bellman-Ford's algorithm. The Bellman-Ford algorithm performs relaxation, gradually decreasing the potential y_v up to the point of reaching the lower bound given by the shortest s-v-path, for each vertex $v \in \mathcal{V}$. In contrast to Bellmann-Ford's arbitrary choice of nodes for relaxation, SPFA utilizes an ordinary queue of vertices for which the potential inequality holds, from which only its corresponding neighbors are considered for relaxation. Despite both algorithms having the same worst case running time, it has been observed that the average running time of the SPFA is $\mathcal{O}(m)$ on random graphs and a strong performance will be confirmed on the performed experiments.

Algorithm	**Case**	**Worst Case Runtime**
Naive Dijkstra	Non-negative arc costs	$\mathcal{O}(n^2)$
Min-Heap Dijkstra	Non-negative arc costs	$\mathcal{O}(m \log n)$
Bellmann-Ford	Arbitrary arc costs	$\mathcal{O}(mn)$
Shortest Path Faster	Arbitrary arc costs	$\mathcal{O}(mn)$

The following listings present the pseudocodes of each algorithm.

Algorithm 3 Naive-Dijkstra($\mathcal{D} = (\mathcal{V}, \mathcal{A})$, s)

```
1: INITIALIZE-SINGLE-SOURCE($\mathcal{D} = (\mathcal{V}, \mathcal{A})$, $s$)
2: $S \leftarrow \mathcal{V}$
3: while $S \neq \emptyset$ do
4:     $v \leftarrow \arg\min_{u \in \mathcal{V}} y_u$
5:     $S \leftarrow S \setminus \{v\}$
6:     for each vertex $w \in Adj[v]$ do
7:         if $y_w > y_v + c_{(v,w)}$ then
8:             RELAX($v, w, c_{(v,w)}$)
9:         end if
10:    end for
11: end while
```

Algorithm 4 Min-Heap-Dijkstra($\mathcal{D} = (\mathcal{V}, \mathcal{A}),\ s$)

```
1:  INITIALIZE-SINGLE-SOURCE($\mathcal{D} = (\mathcal{V}, \mathcal{A}),\ s$)
2:  $S \leftarrow \emptyset$
3:  $Q \leftarrow \mathcal{V}$
4:  while $Q \neq \emptyset$ do
5:      $v \leftarrow$ EXTRACT-MIN($Q$)          ▷ maintain min-heap property after extraction
6:      $S \leftarrow S \cup \{u\}$
7:      for each vertex $w \in Adj[v]$ do
8:          if $y_w > y_v + c_{(v,w)}$ then
9:              RELAX($v, w, c_{(v,w)}$)
10:         end if
11:     end for
12: end while
```

Algorithm 5 Bellmann-Ford($\mathcal{D} = (\mathcal{V}, \mathcal{A}),\ s$)

```
1:  INITIALIZE-SINGLE-SOURCE($\mathcal{D} = (\mathcal{V}, \mathcal{A}),\ s$)
2:  for $i \in 1$ to $|\mathcal{V}| - 1$ do
3:      for each arc $(v, w) \in \mathcal{A}$ do
4:          RELAX($v, w, c_{(v,w)}$)
5:      end for
6:  end for
7:  for each arc $(v, w) \in \mathcal{A}$ do
8:      if $y_w > y_v + c_{(v,w)}$ then
9:          return FALSE
10:     end if
11: end for
12: return TRUE
```

Algorithm 6 Shortest-Paths-Faster($\mathcal{D} = (\mathcal{V}, \mathcal{A}),\ s$)

```
1:  INITIALIZE-SINGLE-SOURCE($\mathcal{D} = (\mathcal{V}, \mathcal{A}),\ s$)
2:  $Q \leftarrow Q \cup \{s\}$
3:  while $Q \neq \emptyset$ do
4:      $v \leftarrow Q$.POP()
5:      for each arc $(v, w) \in \mathcal{A}$ do
6:          if $y_w > y_v + c_{(v,w)}$ then
7:              RELAX($v, w, c_{(v,w)}$)
8:              if $w \notin Q$ then
9:                  $Q \leftarrow Q \cup \{w\}$          ▷ $w$ was previously relaxed
10:             end if
11:         end if
12:     end for
13: end while
```

2.2 Software and implementation

As previously mentioned, the whole project is written on C++, relies solely on the standard library [1]

2.2.1 Design decisions

Throughout the implementation, an object-oriented design was maintained. Despite the possibility of carrying out the implementations with templates and generic programming to allow the instantiation of multiple types, it was concluded that, if the nodes and arcs were given in some non-numeric type, one could always assign indices to their corresponding value in some preprocessing step and perform all operations on such indices. In addition, the goal of this experiment is not to solve an instance of the shortest path problem with specific data, which was another reason not to proceed with this design. However, none of the graphs considered during the experiment contained negative arc costs, such that all algorithms were applicable (even though one can get around negative edge weights in a preprocessing step as it is known from the theory).

In order to facilitate building and compilation, as well as to achieve a clear and understandable interface, the project was divided into libraries, which are linked together with CMake. The listing below represents the directory tree of the project before building and compilation.

Concrete instructions regarding the building and compilation of the project can be found in the README of the repository listed on the footnote.

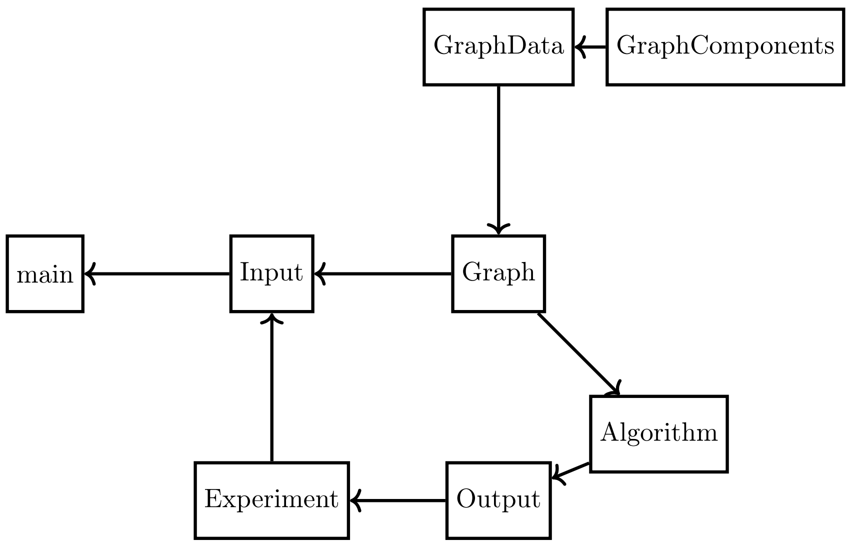
```
./scicomp-projects-g13
    |-- graph_instances
    |-- experiments
    |-- include
    |   |-- SingleSourceShortestPaths
    |       |-- Algorithms
    |       |-- Data
    |       |-- Experiment
    |       |-- Structures
    |-- src
        |-- Algorithms
        |-- Data
        |-- Experiment
        |-- Structures
```

The library **Algorithms** defines a namespace Algorithm in which a ShortestPaths struct and the analyzed algorithms are implemented. I/O-operations and data manipulation are carried out by classes Input, Output and GraphData, which are defined on the **Data** library. Furthermore, the

[1] All code relating to the realization of the project can be found at `https://github.com/TU-SciComp/scicomp-projects-g13`

Experiment library was designed to provide the necessary setting to compute the measurements and present the possibility to perform the experiments and run the individual algorithms, separately, since for instance, the there is no need to provide a sink node to compute the shortest paths tree, whereas for outputing a shortest path computed by a specific algorithm the sink is compulsory. Finally, the **Structures** library holds the interface and implementation of the graph data structure and its respective components.

Below, the inclusion graph of the libraries is provided.



2.3 Implementation

In general, the code is separated into interfaces in header files (.h), and implementations in source files (.cpp). Exceptions to this were template and simple functions, which were directly defined on the headers, and free functions that were right off defined on source files.

In the following, implementation details regarding the algorithms and the graph data structure will be briefly presented.

2.3.1 Graph class

For the implementation of the graph, an adjacency list representation was chosen adapting the source code provided in the lecture. The class utilizes the Node and Neighbor objects defined on the GraphComponents library, where each Node is associated with a vector of Neighbor objects, each holding id and weight as its attributes. Thus, an implicit representation of edges is achieved without having to explicitly implement an edge object.

2.3.2 Algorithms implementations

First, the ShortestPaths data structure will be introduced, as it is passed by reference to every algorithm, in conjunction with a constant reference of a graph object. ShortestPaths holds two vectors that represent both the distances (d) and predecessors (π) associated to every node, which is of practical use to output the results in subsequent computations. Furthermore, the object implements the INITIALIZE-SINGLE-SOURCE and RELAX routines described in the preceding section, where the first is performed during construction, and the latter is called by the four algorithms with different overloads for each of them. Thus, the ShortestPath struct implements all common subroutines among the considered shortest path algorithms, as presented in section 2.1.2

It is worth addressing implementation details and briefly mention minor differences between the pseudocodes and the actual code that were introduced for convenience and to avoid poor performance. First of all, the Q data structure in algorithms 4 and 6 (min-heap and queue, correspondingly) are implemented with a priority queue and an ordinary queue provided by the standard library, respectively. As for differences, in the naive Dijkstra's implementation no actual deletion of a vertex took place as in line 5 of Algorithm 3, but instead, a vector of booleans representing each node was maintained in order to keep track of explored vertices, which requires less memory and computation time than storing and deleting actual elements. A similar approach was used in the implementation of the SPFA.

Another deviation takes place when implementing Algorithm 4 (Min-Heap-Dijkstra), where instead of pushing all nodes to the min-heap at the beginning and relaxing as usual, only the source vertex is inserted at first, and subsequently, the relaxed node is added to the data structure. In this case, no data structure was maintained for a set S either (as in line 2 of Algorithm 4).

For the last two algorithms, the implementation was straightforward and follows the pseudocodes very closely.

2.3.3 Data and experiments

First, all graphs were obtained from `https://www.zib.de/koch/SP/data/` from which only the b15.gph, big.gph, i640-345.gph and world666.gph were subject to the experiments. The table below summarizes characteristics of the graph.

Graph	$\lvert\mathcal{V}\rvert$, $\lvert\mathcal{A}\rvert$	max $\lvert\mathcal{A}\rvert$ given $\lvert\mathcal{V}\rvert$
b15.gph	100, 125	7.750
big8.gph	50.000, 500.000	1.249.975.000
i640-345	640, 40.896	204.480
world666.gph	666, 221.445	221.445

As seen from the table one can claim with certainty that b15.gph and world666.gph are sparse and dense graphs, respectively. On the other hand, the remaining graphs lie somewhere in the middle of this classifications.

In regards to the experiments, the library offers two options to run them: Either give a source node explicitly together with the number of runs that each algorithm is called or give a seed for randomly choosing the source and again, the amount of runs performed. For the sake of simplicity and lack of familiarity to each graph, the experiments were run with the latter option. It is worth highlighting that giving a seed renders the experiments reproducible.

In brief, the experiment consists on running the four algorithms for the number of runs provided and computing the average time accordingly. If the computer has more than four hardware threads, then the algorithms are run in parallel. For each graph, three different seeds were chosen and twenty runs were performed for each algorithm with the exception of big8.gph (five runs were carried out instead) due to the long computation times as a consequence of its magnitude.

3 Results

The outcomes of the aforementioned experiments are summarized in the table below. Detailed output to each individual experiment can be found on the repository under the directory "experimets".

Table 1: Results of experiments.

Graph	**Seed**	**Naive Dijks.**	**Heap Dijks.**	**Bellman-Ford**	**SPFA**
b15.gph	67	4.9340530e-01	6.7041950e-02	4.9407125e-01	6.9640550e-02
	100	1.0511936e+00	4.3598765e-01	1.1353971e+00	4.5254245e-01
	121	1.4029507e+00	7.3274675e-01	1.5262568e+00	5.2562855e-01
big8.gph	4	1.0714370e+05	1.5892000e+02	5.7230851e+05	1.8027233e+02
	7	1.1010706e+05	1.9413595e+02	6.5462434e+05	2.1588852e+02
	10	1.0617931e+05	1.8817119e+02	6.3705959e+05	2.1698025e+02
i640-345	5	1.9953695e+01	3.2504978e+00	6.1046187e+02	2.5960303e+00
	12	2.4523500e+01	3.8557784e+00	6.5588355e+02	5.8743114e+00
	15	2.4365359e+01	3.6975748e+00	6.5383120e+02	5.9700433e+00
world666.gph	5	3.5140496e+01	1.0674370e+01	3.3892749e+03	1.1071532e+01
	13	3.2565494e+01	1.0820079e+01	3.4434747e+03	1.0852752e+01
	67	3.8827676e+01	1.1550349e+01	3.4788404e+03	1.7899010e+01

Table 1: Computation time unit of all algorithms is milliseconds. Seed values were chosen arbitrarily.

Finally, the experiment on graph big8.gph with seed seven was repeated with the -O2 optimization flag to look for improvements in performance. In general, the execution of all algorithms improved by at least one order of magnitude and the file can be found under the "experiments" directory as well.

4 Conclusion

The results observed in the previous section will be briefly analyzed in the following discussion.

As it is to be expected, there is not much variance among individual algorithms with different seeds. However, when comparing different algorithms on the same graph, even in the case of the smallest one, differences in at least one order of magnitude can be observed.

Additionally, a tendency begins to manifest, namely that both the naive variant of Dijkstra and the Bellman-Ford perform significantly more poorly than the Dijkstra's heap implementation and the SPFA. This trend gets solidified as the graphs grow in complexity and furthermore, does not seem to be affected by the two cases of the sparse and dense graphs (b15.gph and world666.gph).

Moreover, as mentioned in section 2.1.3 the SPFA displays a very good performance and its computation time even lies on the same order of magnitude as that of Dijkstra's heap variant, despite having a quadratic worst case running time. Nonetheless, the lack of familiarity with the graphs renders an accurate analysis as to why the SPFA operated so well difficult.

These findings demonstrate that the (worst case) asymptotic runtime of an algorithm does not always accurately reflect its actual computation time in practice and should not solely be used as a guide for deciding to utilize an such algorithm. A deeper analysis of the problem specific problem setting and other factors are required.

References

- Cook, W.J., Cunningham, W.H., Pulleyblank, W.R. and Schrijver, A. (1997). Optimal Trees and Paths. In Combinatorial Optimization (eds W.J. Cook, W.H. Cunningham, W.R. Pulleyblank and A. Schrijver). https://doi.org/10.1002/9781118033142.ch2
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
- https://en.m.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm