Evaluation of algorithms for finding shortest paths in a network

Dani Zugan
89212060@student.upr.si
Faculty of Mathematics, Natural Sciences and Information Technologies
Computer Science
University of Primorska
Glagoljaška 8
SI-6000 Koper, Slovenia

ABSTRACT

The paper evaluates three different algorithms for computing all-pairs shortest paths. We compare the well-known Floyd-Warshall algorithm with two simple modifications of it. The key difference lies in the fact that the relaxations are done in a smarter way. We evaluate the algorithms on three different graph models - uniform Erdős-Rényi, binomial Erdős-Rényi, and Albert-Barabási. Based on the results, we can observe that both modified algorithms outperform the Floyd-Warshall algorithm.

KEYWORDS

Tree algorithm, Hourglass algorithm, Floyd-Warshall algorithm, all-pairs shortest path.

1 INTRODUCTION

Graph theory has long grappled with the timeless challenge of finding the shortest paths within graphs, making it a classic problem in algorithmic studies. The key idea revolves around navigating a (directed) graph, where each arc carries a specific weight, in search of paths that minimize the sum of these arc weights. This fundamental problem finds applications in various real-world scenarios, including bioinformatics, logistics, telecommunications and so on [1].

Two prominent variations of this problem are the single-source shortest path and the all-pairs shortest path (APSP) problems. The single-source variant focuses on discovering paths from a fixed starting vertex to all other vertices in the graph, while the APSP entails finding the shortest path between every possible pair of vertices [4].

In this work, focus will be solely on the APSP variant, aiming to evaluate three different algorithms which offer effective solution. Generally, the APSP problem can be tackled using the technique of relaxation. The core concept of relaxation involves evaluating whether we can enhance the weight of the shortest path from vertex u to v by routing it through vertex w, updating it whenever necessary. Among the well-known relaxation-based solutions, one of the most straightforward approaches is the Floyd-Warshall dynamic programming algorithm which is among algorithms being evaluated. This algorithm boasts a time complexity of $O(n^3)$ when dealing with graphs containing n vertices. While its ease of implementation is commendable, two alternative algorithms will be evaluated which potentially improve efficiency and scalability for solving the APSP problem [4].

In this article, two modifications of the Floyd-Warshall algorithm are introduced and evaluated: the Tree algorithm and the Hourglass algorithm, described in paper Modifications of the Floyd-Warshall algorithm by Andrej Brodnik, Marko Grgurovič and Rok Požar. These modifications offer a notable difference from the original Floyd-Warshall algorithm by implementing a more intelligent approach to carrying out relaxations. This is achieved by introducing a tree-like structure that enables the algorithms to avoid unnecessary relaxations that do not contribute to the final result.

It's important to note that despite these enhancements, both the Tree algorithm and the Hourglass algorithm maintain a worst-case time complexity of $O(n^3)$, which is the same as the classic Floyd-Warshall algorithm. However, their expected performance is significantly improved in practical scenarios [3].

Algortihms are evaluated on graphs generated by using Erdős-Rényi and Albert-Barabási method [2]. Specifically graphs are directed with uniformly distributed arc weights on $[0, 1]$.

2 PRELIMINARIES

A directed graph (digraph), denoted as G, is represented as a pair (V, A), where V is a finite, non-empty set of vertices, and A is a set of arcs that are a subset of $V \times V$. V is denoted as $\{v_1, v_2, ..., v_n\}$ where n is a positive integer greater than or equal to 2 [3].

A path, denoted as P, within a digraph G connecting vertices $v_{P,0}$ to $v_{P,m}$ is a finite sequence of distinct vertices $P = v_{P,0}, v_{P,1}, ..., v_{P,m}$. It is important to note that for $i = 0, 1, ..., m1$, the pair $(v_{P,i}, v_{P,i\ \ 1})$ must represent an arc in the graph G. The length of a path P, represented as |P|, is simply the count of vertices along the path. A k-path refers to a path in which all intermediate vertices are selected from a specific subset $\{v_1, v_2, ..., v_k\}$, where k is a positive integer greater than or equal to 2 [3].

A weighted digraph G is defined as (V, A) with the inclusion of a weight function denoted as w. This function assigns a weight $w(a)$ to each arc a within the set A. In essence, a shortest path from vertex u to vertex v is a path in G whose total weight is the lowest among all possible paths from u to v. The distance between any two vertices, u and v, is defined as the weight of shortest path from u to v within the graph G [3].

3 ALGORITHMS

3.1 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a well-known algorithm which uses simple dynamic programming approach to solve APSP on a weighted, directed graph.The key idea behind the Floyd-Warshall algorithm is that it considers all possible paths between pairs of vertices and gradually refines the shortest distances until it computes the shortest paths for all pairs. Its time complexity is $O(|V|^3)$ due

DOI: https://doi.org/10.26493/scores23.07

to three nested for loops and does not depend on number of arcs. The pseudocode of Floyd-Warshall algorithm is given in Algortihm 1 [3, 4].

1 **for** $k := 1$ *to* n **do**
2   **for** $i := 1$ *to* n **do**
3   **for** $j = 1$ *to* n **do**
4   **if** $W_{ik} \quad W_{kj} < W_{ij}$ **then**
5                 $W_{ij} := W_{ik} \quad W_{kj}$;
6   **end**
7   **end**
8   **end**
9 **end**

Algorithm 1: Floyd-Warshall(W)

3.2 The Tree algorithm

The Tree algorithm is the first version of the modified Floyd-Warshall algorithm. Consider the k-th iteration, and let OUT_k represent a shortest path tree originating from vertex v_k. It is based on the observation that the relaxation in lines 4-5 would not always succeed in lowering the value of W_{ij}. Instead of simply looping through every vertex of V in line 3, we perform the depth-first traversal of OUT_k. This allows us to skip iterations where it's guaranteed that they won't reduce the current value of W_{ij} [3].

The pseudocode for the enhanced algorithm, known as the Tree algorithm, can be found in Algorithm 2. The first step involves the construction of the tree OUT_k using the ConstructOUT procedure outlined in Algorithm 3. Subsequently, a depth-first search operation is carried out [3].

1 Initialize π, an $n \times n$ matrix, as $\pi_{ij} := i$;
2 **for** $k := 1$ *to* n **do**
3     $OUT_k := ConstructOUT_k(\pi)$;
4   **for** $i := 1$ *to* n **do**
5   Stack := empty;
6   Stack.push(v_k);
7   **while** *Stack*   *empty* **do**
8   V_k := Stack.pop();
9   **forall** *children* v_j *of* v_x *in* OUT_k **do**
10   **if** $W_{ik} \quad W_{kj} < W_{ij}$ **then**
11                     $W_{ij} := W_{kj} + W_{kj}$;
12                     $\pi_{ij} := \pi_{kj}$;
13   Stack.push(v_j);
14   **end**
15   **end**
16   **end**
17   **end**
18 **end**

Algorithm 2: Tree(W)

In Algorithm 2 vertices of OUT_k are visited in DFS order, which is facilitated by using the stack. We can avoid pushing and poping of

1 Initialize n empty trees: $T_1, T_2, ..., T_n$.;
2 **for** $k := 1$ *to* n **do**
3     $T_1.Root := v_1$;
4 **end**
5 **for** $i := 1$ *to* n **do**
6   **if** $i \quad k$ **then**
7   Make T_i a subtree of the root of T_{k_i}.
8   **end**
9 **end**
10 **return** T_k

Algorithm 3: $ConstructOUT_k(\pi)$

each vertex by precomputing two read-only arrays dfs and $skip$ to support the traversal of OUT_k. The array dfs comprises the vertices of OUT_k that have been visited during the depth-first search (DFS) traversal. In contrast, the array $skip$ serves the purpose of bypassing the subtree of OUT_k when relaxation of edges in that subtree does not yield successful results [3].

3.3 The Hourglass algorithm

The Tree algorithm can be further improved by using another tree. The second tree, represented as IN_k, resembles a shortest path tree, with the difference being that it serves as a shortest path "graph" for routes from v_i to v_k for every $v_i \in V$ except v_k. Precisely IN_k is not a tree, but if direction of arcs is reversed, it shifts it into a tree with v_k as the root. This is actually a substitute of the for loop on variable i in line 2 of Algorithm 1 and in line 4 of Algorithm 2. This modification of Floyd-Warshall algorithm is named the Hourglass algorithm, the name comes from placing IN_k tree atop the OUT_k tree, which gives it an hourglass-like shape, with v_k being at the neck. The pseudocode of the Hourglass algorithm is given in Algorithms 4 and 5. Additional algorithm constructs IN_k similarly to the construction of OUT_k, except that the matrix ϕ_{ik} is used instead [3].

1 Initialize π, an $n \times n$ matrix, as $\pi_{ij} := i$;
2 Initialize ϕ, an $n \times n$ matrix, as $\phi_{ij} := i$;
3 **for** $k := 1$ *to* n **do**
4     $OUT_k := ConstructOUT_k(\pi)$;
5     $IN_k := ConstructIN_k(\phi)$ **forall** *children* v_i *of* v_k *in* IN_k **do**
6   RecurseIN(W, π, IN_k, OUT_K, v_i);
7   **end**
8 **end**

Algorithm 4: $Hourglass(W)$

In practice, the algorithm's efficiency can be enhanced by employing an additional stack instead of recursion. This optimization significantly speeds up the implementation process. It's important to highlight that the worst-case time complexity of the Hourglass and Tree algorithm stays at $O(n^3)$. This scenario is evident when all shortest paths are direct arcs themselves, resulting in a tree structure where all leaves are children of the root, and this configuration remains unchanged throughout the algorithm's execution [3].

```
1  Stack := empty;
2  Stack.push(v_k);
3  while Stack   empty do
4      v_x := Stack.pop();
5      forall children v_j of v_x in OUT_k do
6          if W_ik   W_kj < W_ij then
7              W_ij := W_ik   W_kj;
8              π_ij := π_kj;
9              ϕ_ij := ϕ_ik;
10             Stack.push(v_j);
11         else
12             Remove the subtree of v_j from OUT_k;
13         end
14     end
15 end
16 forall children v_i' of v_i in IN_k do
17     RecurseIN(W, π, ϕ, IN_k, OUT_K, v_i');
18 end
19 Restore OUT_k by reverting changes done by all iterations of
     line 12;
```

Algorithm 5: *RecurseIN*$(W, \pi, IN_k, OUT_K, v_i)$

4 EVALUATION

4.1 Testing environment

All algortihms and graph generators were implented in C and C++ and compiled using gcc version 6.3.0. The tests were ran on an AMD Ryzen 7 5700U with 16GB RAM running on Windows 11 64-bit.

4.2 Graph generation

Three different variant of random graphs were generated using Erdős-Rényi model and Albert-Barabási model[1]. At the beginning of generating each variant seed was set to 1 and was incremented by +1 for each new graph. All graphs generated underwent a DFS search to ensure strong connectivity[2].

Firstly, using binomial Erdős-Rényi model the input values to random graphs were the number of vertices, denoted as n, and probability, denoted as p. Each possible arc $(n * (n-1))$ in a directed graph is included with probability p, independently from every other arc. Weights are added uniformly from the interval [0,1]. Four different sizes of graphs (512, 1024, 2048 and 4096 vertices) and five different probabilities (0.1, 0.3, 0.5, 0.7, 0.9) were selected as input. For each instance of the input five different graphs were created, all together 100 different graphs were created using this model.

Secondly, a different variant was used, called uniform Erdős-Rényi model. Input values for creating graphs were number of vertices and number of arcs, denoted by m. Out of all $n * (n-1)$ possible arcs in a directed graph a random permutation was made to select the desired number of arcs. Weights are added uniformly from the interval [0,1]. Again the sizes of graphs were 512, 1024, 2048 and 4096 vertices and inputs for m were $5 * n$, $10 * n$, $20 * n$, $50 * n$, $100 * n$. All together 100 different graphs were created using this model.

Finally, using Albert-Barabási model the input values for creating random graphs were number of vertices of final graph, number of vertices of initial graph, denoted by c, and number of arcs added in each round, denoted by m. Initially a clique of size c is created. At each step, one new vertice is added, with m new arcs to the vertices already in the graph. With preferential attachment the vertices with higher degree have a higher probability to receive new arcs. The graph constructed after $n - c$ steps is undirected. To determine the direction of each arc, a function similar to a coin flip is employed. This function randomizes the orientation of the arcs, ensuring an equal distribution where half of the arcs lie above the main diagonal in the adjacency matrix, and the other half lie below it. For this evaluation the m value was fixed at 15 and C value was fixed at 30. Again the sizes of graphs were 512, 1024, 2048 and 4096 vertices and for each instance 20 different graphs were generated (80 together). Weights are added uniformly from the interval [0,1].

4.3 Evaluation

As mentioned, Tree and Hourglass algorithm were compared with Floyd-Warshall algorithm. The results on graphs generated by binomial Erdős-Rényi model are shown in figure 1, by uniform Erdős-Rényi model are shown in figure 2 and by Albert-Barabási model are shown in figure 3. To better visualize the results, natural logarithm of arcs is used on the x axis. Both modifications performed much better than Floyd-Warshall algorithm, especially as the number of vertices gets higher. It is worth noting that, between the Tree and Hourglass algorithms, the Tree algorithm demonstrated slightly better results in the performance comparison.

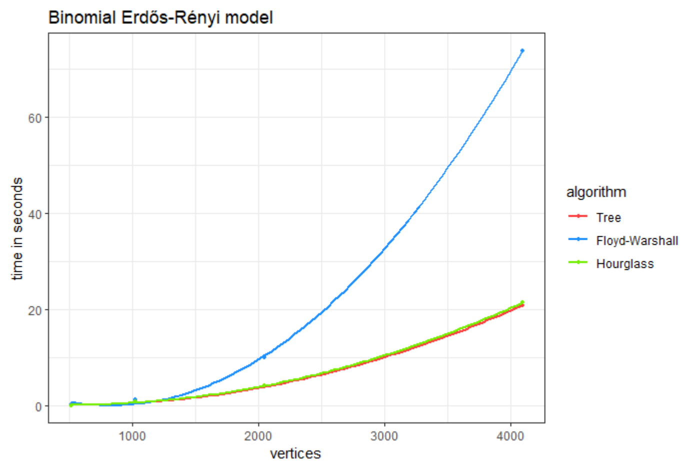

Figure 1: Binomial Erdős-Rényi model results

In figure 4 results of all three variants of graphs are shown. As expected both modifications were slower as the graph got more dense and the number of vertices stayed the same. Moreover also Floyd-Warshall algorithm was a little bit slower as graph got denser which is unexpected.

[1] Link to code for generating random graphs is available here: GitHub Repository
[2] In Albert-Barabási model some seeds did not produce strongly connected graph and therefore more than 80 seeds were used.

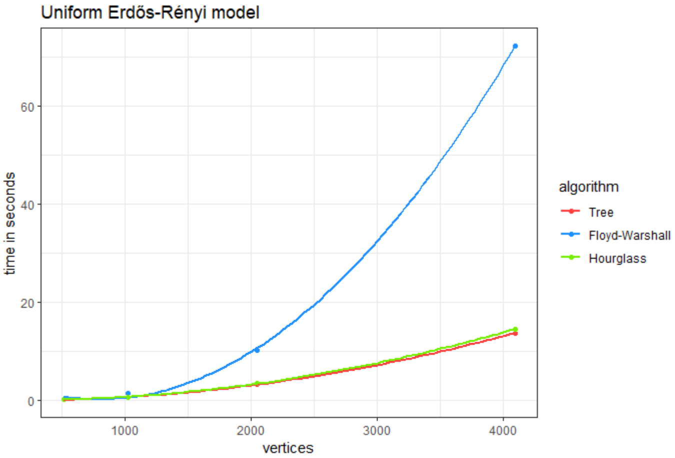

Figure 2: Uniform Erdős-Rényi model results

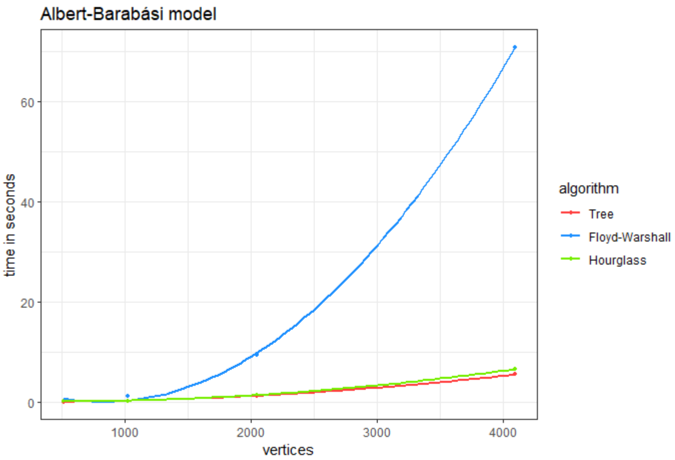

Figure 3: Albert-Barabási model results

5 CONCLUSIONS

In this paper, a straightforward evaluation of three algorithms for finding shortest paths in a network was presented. Random networks were generated using the Erdős-Rényi and Albert-Barabási models. The paper's results showed that both modifications of the algorithms performed better than the Floyd-Warshall algorithm, especially when the size of the graphs (vertices) was larger.

To further enhance the comprehensiveness of the evaluation, the Tree and the Hourglass algorithms will be further assessed on graphs that are not strongly connected. By incorporating non-strongly connected graphs into the assessment, deeper insights into the behavior and robustness of the algorithms in real-world scenarios, are expected to be gained.

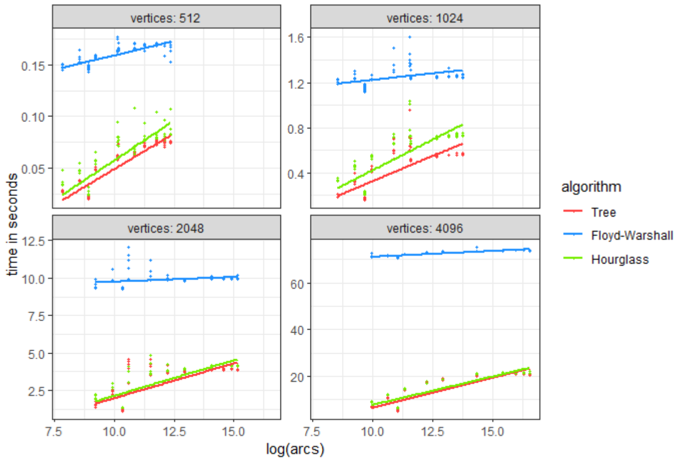

Figure 4: Combined results

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., USA.
[2] Albert-Laszlo Barabasi and Rita Albert. 1999. Albert, R.: Emergence of Scaling in Random Networks. Science 286, 509-512. *Science (New York, N.Y.)* 286 (11 1999), 509–12. https://doi.org/10.1126/science.286.5439.509
[3] A. Brodnik, M. Grgurovič, and R. Požar. 2021. Modifications of the Floyd-Warshall Algorithm with Nearly Quadratic Expected-Time. *ARS MATHEMATICA CONTEMPORANEA* (2021). https://amc-journal.eu/index.php/amc/article/view/2467
[4] R. W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5 (1962), 345. https://doi.org/10.1145/367766.368168