The 2^{nd}-shortest Path

Project 3 Report
Fundamental of Data Structure (2023 Fall)
Zhejiang University BY

Author Name

Student ID:
Teacher: HongHua Gan

Accomplished on December 2, 2023

Made
With LaTeX

Declaration

I hereby declare that all the work done in this project titled “The 2$^{\text{nd}}$-shortest Path” is of my independent effort.

Contents

Chapter 1

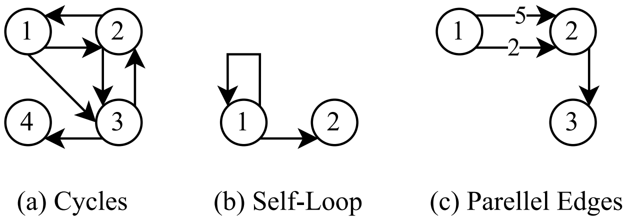
Introduction

1.1 Problem Description

Give a **directed** graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge $e \in E$ has a non-negative weight $w(e)$. Given two vertices s and t, find the 2^{nd}-shortest path from s to t.

The graph **can** contain **cycles, self-loops and parellel edges**. For example:

Figure 1.1: Structures in the graph



Note that the **end vertex** cannot have any outgoing edges. As TA said:

> Although the question does not say it clearly, according to the meaning of the question, Lisa wants to go home, but it does not follow the shortest path but the shortest path, so once she reaches the end station M, it should be considered that she is going home.

If there are multiple 2^{nd}-shortest paths, we can output any of them. If there is no 2^{nd}-shortest path, should inform the user.

1.2 Problem Background

This problem is a generalization of the shortest path problem. The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. The best known algorithm for this problem is Dijkstra's algorithm, which runs in $O(|E| + |V| \log |V|)$ time. The algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

We consider a long-studied generalization of the shortest path problem, in which not one but several short paths must be produced. The k shortest paths problem is to list the k paths connecting a given source-destination pair in the digraph with minimum total length. Our techniques also apply to the problem of listing all paths shorter than some given threshhold length. In the version of these problems studied here, cycles of repeated vertices are allowed. We first present a basic version of our algorithm, which is simple enough to be suitable for practical implementation while losing only a logarithmic factor in time complexity. We then show how to achieve optimal time (constant time per path once a shortest path tree has been computed) by applying Frederickson's algorithm for finding the minimum k elements in a heap-ordered tree.

The 2^{nd}-shortest path problem is a simpler version of the k shortest path problem. Because you just care about the 2^{nd}-shortest path, you don't need to maintain a list and find the k shortest paths.

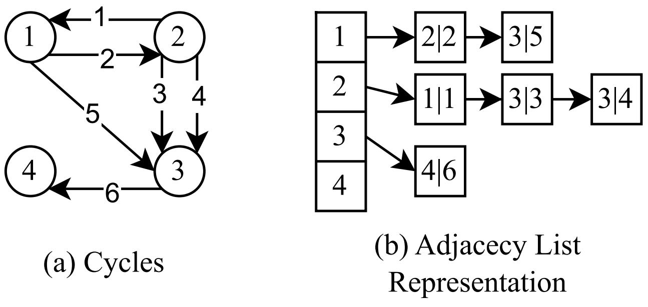
Chapter 2

Algorithm Specfication

2.1 Data Structure

I use adjacency list to represent the graph. The graph is stored in a `Graph` struct. The `Graph` struct contains a `Vertex` pointer array `vertices` and a `numVertices` field. The `Vertex` struct contains two `int` fields `dest` and `distance`, representing one outgoing edge and its weight. The `Vertex` struct also contains a `Vertex` pointer `next`, representing the next vertex in the adjacency list.

Figure 2.1: Data structure of the graph



2.2 Algorithm

I use breadth-first search (BFS) to find the shortest path from s to t. The algorithm is shown below:

Algorithm 1: BFS Algorithm

```
Input: Path $P$ from $s$ to $q_i$
Output: Pathes $P'_k$ from $s$ to $q_{i+1}$ that are derived from $P$
1 if $q_i$ has no outgoing edges then
2     return NULL
3 end
4 $P' \leftarrow P$
5 forall $e \in E(q_i)$ do
6     $P' \leftarrow P' + e$
7 end
8 return $P'_k$
```

All the pathes that haven't reached t are stored in a priority queue, so BFS will always pop the current shortest path from the queue and explore it. If the path reaches t, we need to determine whether it is the 2nd-shortest path. If it is, we output it. If it is not, we continue to explore the next shortest path.

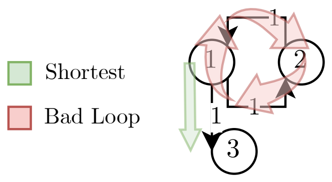
Algorithm 2: 2-nd Selection Algorithm

```
Input: Graph $G$, start vertex $s$, end vertex $t$
Output: The 2th-shortest path from $s$ to $t$
1  $Q \leftarrow \emptyset$
2  $Q.push(s)$
3  while $Q$ is not empty do
4      $P \leftarrow Q.pop()$
5      if $P$ reaches $t$ then
6          if $P$ is the 2th-shortest path then
7              return $P$
8          else
9              continue
10         end
11     end
12     forall $P' \in BFS(P)$ do
13         $Q.push(P')$
14     end
15 end
16 return $NULL$
```

To determine whether a path is the 2nd-shortest path is simple. We just need to compare the length of the path with the length of the shortest path. When we first encountered t, the path is the shortest path. Record this length. When we encountered t again, the path is the 2nd-shortest path if and only if the length of the path is not equal to the length of the shortest path.

It is worth noting that without the existence of a sub-shortest path, breadth-first search may get stuck in a loop, causing the algorithm's loop condition `Q.isNotEmpty()` to never terminate, as shown in the diagram below. Therefore, when stepping through each path, it should be checked whether it has become trapped in a loop. Below, I will provide a proof that if a path passes through the same node more than twice, then it is definitely not a 2nd path. For such paths, they can be directly discarded.

Figure 2.2: Bad Loop



Theorem 2.2.1. *Let P be a path containing vertices $v_1, v_2, \ldots, v_n$. If P passes through v_i more than twice, then P is not a 2^{nd} path.*

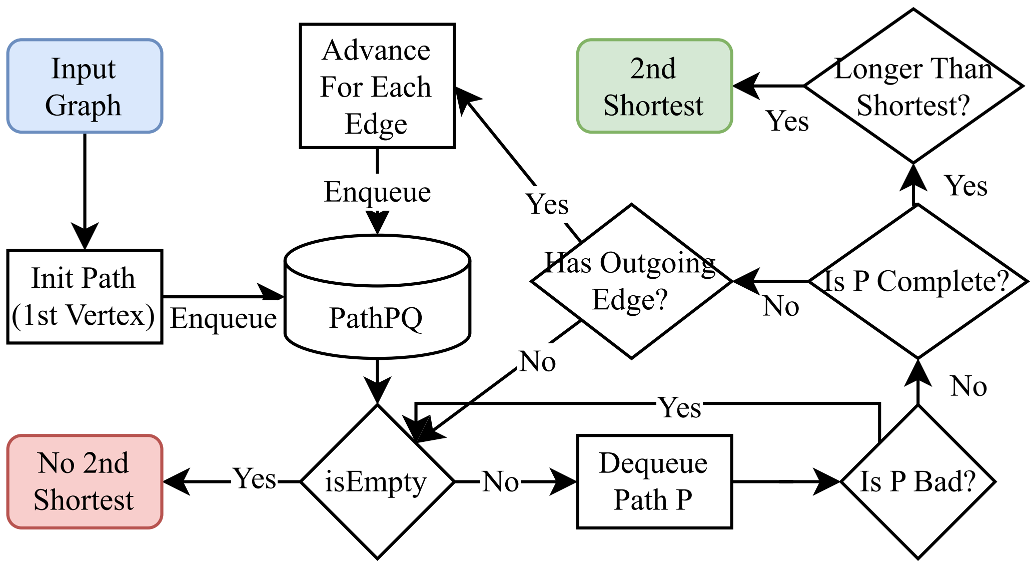
Proof. Starting from the first appearance of v_i, we can divide P into parts $P_1, P_2, \ldots, P_k$, where P_1 is the path from the start vertex to the first appearance of v_i, P_k is the path from the last appearance of v_i to the end vertex, and P_j is the path from the $(j-1)$th appearance of v_i to the jth appearance of v_i. Obviously, $P_1, P_2, \ldots, P_k$ are all loops that pass through v_i. Each loop P_j have a length of at least 0, and there must be a smallest loop P_s. If a path passes through v_i, then the 2nd shortest path must passes through P_s and then never passes through v_i again (otherwise it will form a longer loop). Therefore, if a path passes through v_i more than twice, then it is definitely not a 2nd path. ■

Actually, by induction, we can prove that if a path passes through the same node more than k times, then it is definitely not a kth path. The proof is left to the reader.

2.3 Program Architecture

The picture below shows the architecture of the program. The input will be parsed into a graph. Then the program will run the 2nd-shortest path algorithm relying the `PathPQ` priority queue. Finally, the program will output the result.

Figure 2.3: Sketch of the program



Chapter 3

Test Cases

3.1 Table of Test Cases

I've prepared a complete test suite for the program in `code/testdata`. Some of the test cases are shown in Table ??.

Table 3.1: Test Results

Begin of Table				
No.	Purpose	Input	Output	Correctness

1	`pta.in`
	PTA provided test
		`5 6`
		`1 2 50`
		`...`
			`240 1 2 4 5`	Yes

2	`2.in`
	Simple test 2
		`6 9`
		`1 2 10`
		`...`
			`7 1 5 3 4 2 6`	Yes

3	`3.in`
	Simple test 3
		`9 20`
		`1 4 546`
		`...`
			`1015 1 4 7 9`	Yes

4	`loop1.in`
	Single loop test
		`5 5`
		`1 2 1`
		`...`
			`10 1 2 3 4 2 3 5`	Yes

5	`loop2.in`
	(comprehensive)
	Multiple loop test
		`6 7`
		`1 2 1`
		`...`
			`13 1 2 3 5 2 3 6`	Yes

6	`extreme.in`
	(largest)
	Extreme test
		`1000 5000`
		`1 2 2`
		`...`
		`977 989 6`
			`63 1 95 216 556 1000`	Yes

7	`no_2nd.in`
	No 2nd-shortest path test
		`3 2`
		`1 2 1`
		`2 3 2`
			`[Error] No second shortest path.`	Yes

8	`one.in` (smallest) One node test	`1 0`	`[Error] No second shortest path.`	Yes

9	`self.in`
	(extreme)
	Self-loop test
		`3 5`
		`1 1 1`
		`...`
			`3 1 1 2 3`	Yes

10	`para.in`
	(extreme)
	Parallel edge test
		`2 2`
		`1 2 1`
		`1 2 2`
			`2 1 2`	Yes

11	`mid.in`
	Middle vertex test
		`100 200`
		`1 2 9`
		`...`
			`68 1 7 8 36 70 100`	Yes

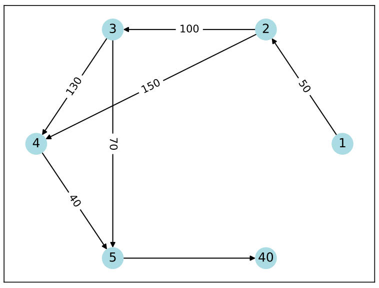
End of Table				

3.2 Visualization

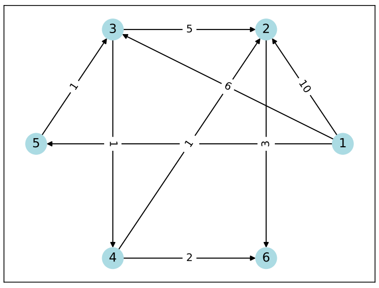
Their visualized graphs are shown in Figure 3.1.

Figure 3.1: Test Graphs

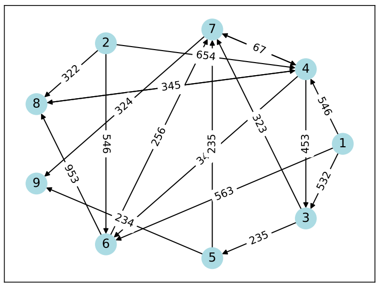
(a) pta.in



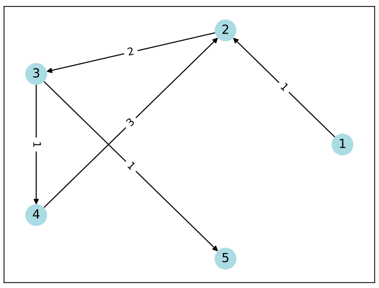
(b) 2.in



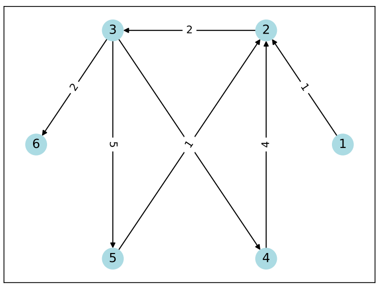
(c) 3.in



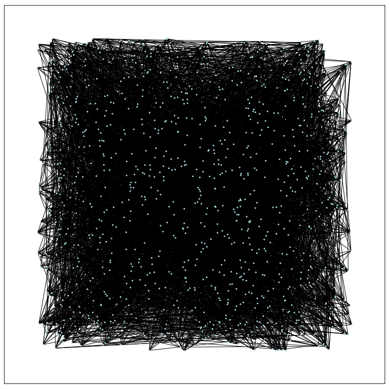
(d) loop1.in



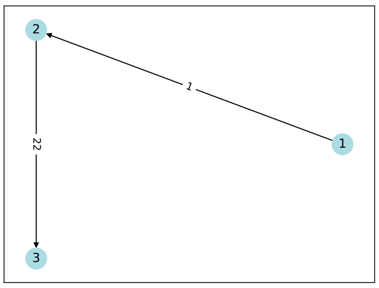
(e) loop2.in



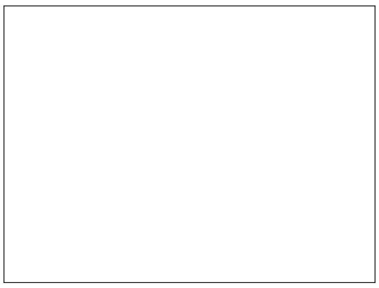
(f) extreme.in



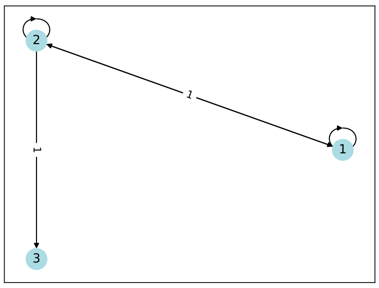
(g) no_2nd.in



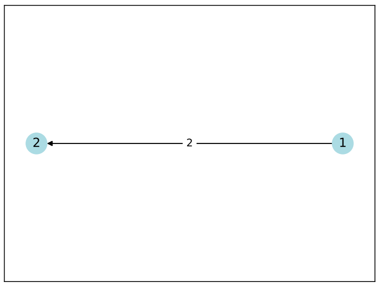
(h) one.in



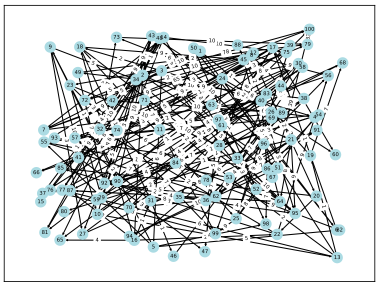
(i) self.in



(j) para.in



(k) mid.in



Chapter 4

Analysis and Comments

4.1 Time Complexity

Theorem 4.1.1. *The time complexity of the algorithm is* $O(|E| + |V| \log |V|)$.

Proof. Analyse the time complexity of the algorithm step by step.

Step I Construct the graph. The time complexity is $O(|E|)$.

Step II Run BFS until the shortest path is found. This path must be a loopless path. The time complexity is $O(|V|)$.

Step III The algorithm will run in two cases.

Case 1 Run BFS until the 2$^{\text{nd}}$-shortest path is found. As shown Theorem 2.2.1, the 2$^{\text{nd}}$-shortest path can contain at most one loop, so the time complexity is $O(|V|)$. Before this path is found, the algorithm will sort the pathes in the priority queue. The time complexity is $O(|V| \log |V|)$.

Case 2 If no 2$^{\text{nd}}$-shortest path is found, the algorithm will traverse all the pathes (containing at most one loop) in the priority queue. The time complexity is $O(|E|)$.

Step IV Output the result. The time complexity is $O(|V|)$.

Therefore, the time complexity of the algorithm is $O(|E| + |V| \log |V|)$. ■

4.2 Space Complexity

Theorem 4.2.1. *The space complexity of the algorithm is* $O(|E| + |V|)$.

Proof. The space complexity of the algorithm is the sum of the space complexity of the graph and the priority queue.

The space complexity of the graph is $O(|E| + |V|)$.

The space complexity of the priority queue is complicated. The priority queue contains all the pathes that haven't reached t. Because pathes with more than one loop are discarded, each path contains at most $2|E|$ vertices. The number of pathes in the priority queue is at most $|V|$ (BFS). Therefore, the space complexity of the priority queue is $O(|E| + |V|)$.

In total, the space complexity of the algorithm is $O(|E| + |V|)$. ■

4.3 Comments

Many papers study algorithms for k shortest paths [1]. Dreyfus and Yen [2] cite several additional papers on the subject going back as far as 1957.

One must distinguish several common variations of the problem. In many of the papers cited above, the paths are restricted to be simple, i.e. no vertex can be repeated. This has advantages in some applications, but as our results show this restriction seems to make the problem significantly harder. Several papers [1] consider the version of the k shortest paths problem in which repeated vertices are allowed, and it is this version that we also study. Of course, for the DAGs that arise in many of the applications described above including scheduling and dynamic programming, no path can have a repeated vertex and the two versions of the problem become equivalent. Note also that in the application described earlier of listing the most likely failure paths of a system modelled by a finite state machine, it is the version studied here rather than the more common simple path version that one wants to solve.

Appendix A

Source Code

A.1 Source Code Statistics

The source code is in the `code` directory. I've commented the source code using Doxygen-style comments.

1. `main.c`: Driver program.
2. `graph.c, graph.h`: Implementation of the expression tree.

And here is the statistics of the source code above using `cloc` tool:

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                2             38            201            289
C/C++ Header                     1              8             21             45
-------------------------------------------------------------------------------
SUM:                             3             46            222            334
-------------------------------------------------------------------------------
```

39% of the source code is well documented with Doxygen-style comments. This satisfies the requirement of the project.

A.2 Compilation and Execution

The program is written in C and can be compiled using `gcc`. There is a `Makefile` in the `code` directory. If you use UNIX system, you can compile the program by running `make` in the `code` directory.

If you use Windows system, you can compile the program by running the following command in the `code` directory:

```
gcc -o main.exe main.c graph.c
```

If you use IDE, you can also import the source code into the IDE and compile it.

After compilation, you can run the program by running `main` or `main.exe` in the `code` directory.

The program will recognize the input from command line. If you want to input the expression from a file, you can run the program like this:

```
./main input.txt
```

If you don't specify the input file, the program will read the expression from standard input. You can redirect the input from a file like this:

```
./main < input.txt
```

Or you can type the graph in the terminal.

I've prepared a complete test suite for the program in `code/testdata`. If you use UNIX system, you can run the program on the test data by running the following command in the `code` directory:

```
make test
```

It will run the program on all the test data and save the output in `code/testdata` with `.out` suffix.

If you don't use UNIX system, you can test the program manually:

```
./main testdata/pta.in
```

Bibliography

[1] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[2] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.