

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



Discrete Mathematics-CO1007

Assignment's Report

Lecturer: Trần Tuấn Anh
Class: CC02
Group: 02
Students: Cao Minh Quang - 2052221

HO CHI MINH CITY, 11th DECEMBER 2021



Contents

1	Introduction	2
2	Notation and Definitions	2
3	Description of Yen's algorithm	3
4	The pseudo code for Yen's algorithm	4
5	Features of Yen's algorithm	5
6	Example for the usage of Yen's algorithm	5
6.1	Example 1:	5
6.2	Example 2:	8
6.3	Example 3:	12
7	Recheck the results with JavaScript Program	19
7.1	Example 1:	20
7.2	Example 2:	23
7.3	Example 3:	25
8	Improvements for Yen's algorithm	26
	References	28

1 Introduction

When discussing about the k^{th} shortest paths problem, there are two main types. The first one is to find out k shortest paths from the source to the destination in which **loops are allowed**. The second one has the same requirement but **no loops are allowed**.

In this part, we focus on the second type only. There are many algorithm have been proposed to tackle this problem by Bock, Kantner and Haynes, Pollack, Clarke, Krikorian, and Rausan, Sakarovitch...

However, the algorithm published by Jin Y. Yen in 1971 is still by far more efficient to compute single source K -shortest loop-less paths for a graph with real-valued positive weight.

In Yen's algorithm, the shortest path is computed first using Dijkstra algorithm, then each vertex of this shortest path is used to obtain another shortest path to the destination. This principle is successively applied on each vertex of previously obtained paths. When computing K -shortest paths using this method, some vertices and edges need to be removed from the graph to find different paths and are ignored by the search by tapping directly into Dijkstra algorithm.

The significance of Yen's algorithm is that its computational upper bound increases only linearly with the value of K .

2 Notation and Definitions

In a graph with N vertices, let:

- (i), a vertex of the graph where $i = 1, 2, \dots, N$. This means that (1) is the source and (N) is the destination.
- The notation $(1) - (i) \dots - (j)$, $i \neq j \neq \dots \neq 1$ represents the path from (1) to (j) passing through (i).
- d_{ij} , be the weight of the edge connecting (i) and (j). If this edge exists, d_{ij} is a finite number, otherwise, d_{ij} is considered equal to infinity.
- $A^k = (1) - (2^k) - (3^k) - \dots - (Q_k^k) - (N)$, $k = 1, 2, \dots, K$, be the k^{th} shortest path from (1) to (N) where (2^k) , (3^k) , ..., (Q_k^k) are the 2^{nd} , 3^{rd} , ..., Q_k^{th} vertex of the k^{th} shortest path.
- A_i^k with $i = 1, 2, \dots, Q_k$ be a **set of deviations from A^{k-1} at (i)**, a deviation from A^{k-1} at (i) is the shortest of the paths that coincide with A^{k-1} from (1) to the i^{th} vertex and then deviate to a vertex that is different from any of the $(i+1)^{\text{st}}$ vertex of those A^j , $j = 1, 2, \dots, k-1$.

In the order words, the set A_i^k has the same path with A^{k-1} from (1) to the i^{th} vertex, but reaches (N) by a shortest sub-path without passing any vertices that is already included in the first part of the path. **Note that A_i^k is loop-less and contains the same vertex no more than once.**

- R_i^k be the root of A_i^k , the root of A_i^k is the sub-path of A_i^k that coincides with A^{k-1} , i.e., $(1) - (2^k) - \dots - (i^k)$ in A_i^k .
- S_i^k be the spur of A_i^k , the spur of A_i^k is the last part of A_i^k that has only one vertex coinciding with A^{k-1} , i.e., $(i^k) - \dots - (N)$ in A_i^k .

3 Description of Yen's algorithm

* The algorithm can be broken down into 2 procedure:

– **Iteration 1: Determining A^1**

The Dijkstra algorithm is used first to find the shortest path A^1 from the source to the destination.

When there are negative loops in the graph, this K-shortest paths algorithm has to be terminated. However, when there are no negative loops in the graph, we should obtain at least one path that has the shortest length. If we have K or more such paths, we are done. If we have less than K and more than one paths, we assign any arbitrary one of these paths to be A^1 and store it in List A (the list of k-shortest paths), the rest of these paths are stored in List B (the list of candidates for $(k+1)^{st}$ shortest paths). Otherwise, if we have only one such path, it is A^1 which is to be stored in List A.

– **Iteration k ($k = 2, 3, \dots, K$): Determining A^k**

In order to find A^k , the shortest paths A^1, A^2, \dots, A^{k-1} must have been found previously. Then A^k is determined as follows:

a. For each of $i = 1, 2, \dots, Q_{k-1}$, do the following steps:

Step 1: Check if the sub-path consisting of the first i vertices of A^{k-1} in sequence coincide with the sub-path consisting of the first i vertices of A^j in the sequence for $j = 1, 2, \dots, k-1$. If so, set $d_{iq} = \infty$ where (q) is the $(i+1)^{st}$ vertex of A^j ; otherwise, make no changes. Then move to the next step. (Note that d_{iq} are set to ∞ for computation in iteration k only. They will be restored to their original values before iteration k+1 starts.)

Step 2: Apply the Dijkstra algorithm to find the shortest path from (i) to (N) , allowing it to pass through vertices that are not yet included in the path. (Note that the sub-path from (1) to (i) is R_i^k , the root of A_i^k , and the sub-path from (i) to (N) is S_i^k , the spur of A_i^k . And also if there are more than one sub-paths from (i) to (N) that have the minimum length, take any arbitrary one of them.)

Step 3: Find A_i^k by joining R_i^k and S_i^k . Then add A_i^k to List B.

b. Find from List B the path(s) that have minimum length

If the path(s) found plus the path(s) already in List A exceed K, we are done. Otherwise, denote this path (or an arbitrary one, if there are more than one such paths) by A^k and move it from List B to List A, leaving alone the rest of the paths in List B. Then go to iteration k+1.

* **Some notification:**

A^k is a deviation from A^j , $j = 1, 2, \dots, k-1$. More precisely, A^k must coincide with A^j , $j = 1, 2, \dots, k-1$ for the first $n \geq 1$ vertices then deviates to a different vertex and finally arrives at the destination without passing each vertex more than once.

Therefore, to obtain A^k it is only necessary to look for all shortest deviations from the $(A^j)'$ s, then scan from these deviations the one that has the shortest length.

In iteration k, step 1 of part a, setting d_{iq} equal to ∞ to force A^{k-1} to deviate at each vertex on the path without allowing the deviation to take any path that have shorter length than A^{k-1} . This is followed by step 3 of part a which find the shortest deviations of A^{k-1} that are different from A^j , $j = 1, 2, \dots, k-1$.

Finally, in part b, the A^k is selected from all possible paths in List B. Therefore the $A^j, j = 1, 2, \dots, K$, thus obtained by the iterative procedure are the K shortest loop-less paths from the source to the destination.

4 The pseudo code for Yen's algorithm

```
procedure Yen_shortest_loopless(G, source, destination, K):
```

```
    A[1] := Dijkstra(G, source, destination);
```

```
    B := []; //Initialize List B
```

```
    for k from 2 to K:
```

```
        for i from 1 to size(A[k-1])-1: //Spur node is scanned from the 1st vertex to
            the one next to the last vertex of the previous k-shortest path
```

```
            spur_node := A[k-1].node(i); //Obtain spur node from the previous k-shortest
            path
```

```
            root_path := A[k-1].nodes(1, i); //The root path is the coincident path from
            node 1 to node i
```

```
            for each path p in A:
```

```
                if root_path = p.nodes(1, i):
```

```
                    remove p.edge(i, i + 1) from G; //Remove the edge that are part of the
                    previous shortest paths sharing the same root path
```

```
            for each root_path_node in root_path except spur_node:
```

```
                remove root_path_node from G;
```

```
            spur_path = Dijkstra(G, spur_node, destination); //Checking if any spur path
            found
```

```
            total_path = root_path + spur_path;
```

```
            if (total_path not in B):
```

```
                B := B + (total_path); //Add the k-shortest path in B
```

```
            restore edges to G;
```

```
            restore nodes in root_path to G; //Add back the edges and vertices that were
            removed from the graph
```

```
        if B is empty:
```

```
            break;
```

```
        // This is for the case of no spur paths found or no spur path left
```

```
        B.sort(); //Sort the k-shortest paths by weight
```

```
        A[k] = B[1]; // Add lowest weight path to List A
```

```
return A;
```

5 Features of Yen's algorithm

Time complexity:

The time complexity of Yen's algorithm depends on the Dijkstra's algorithm. Dijkstra has a time complexity of $O(N^2)$, but using a Fibonacci heap make it become $O(M + N \log N)$ where M is the amount of edges in the graph. Since Yen's algorithm makes $K.l$ calls to the Dijkstra in computing the spur path, where l is the length of spur paths. In a condensed graph, the expected value of l is $O(\log N)$, while the worst case is N , the time complexity then becomes $O[KN(M + N \log N)]$.

Space complexity:

Storing the edges of the graph in the List A and List B, are totally required $N^2 + K.N$ memory addresses. The worse case would be every vertex in the graph has an edge to every other vertex in the graph, then N^2 addresses are needed. Normally, only $K.N$ addresses are used for both list A and B since at most only K paths will be stored, where it is possible for each path to have N vertices.

6 Example for the usage of Yen's algorithm

6.1 Example 1:

Find top 3 shortest path from C to H without any loops in the following graph:

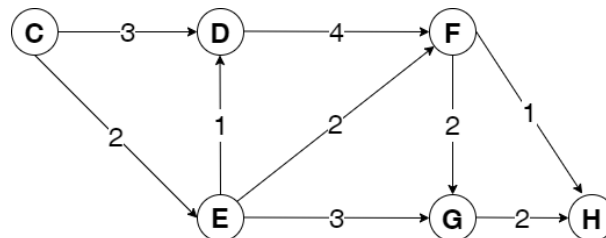


Figure 1: The given graph.

- Firstly, using Dijkstra's algorithm to find the shortest path A^1 from (C) to (H) in the graph.

S	C	D	E	F	G	H
\emptyset	0	∞	∞	∞	∞	∞
C	0	3	2	∞	∞	∞
E	0	3	2	4	5	∞
D	0	3	2	4	5	∞
F	0	3	2	4	5	5
G	0	3	2	4	5	5

We have: $A^1 = (C) - (E) - (F) - (H)$ with the total weight of 5. This path is appended to List A and becomes the first 3-shortest path.

- Secondly, we are going to find A^2 :
 - Vertex (C) of A^1 becomes the spur vertex with $R_1^2 = (C)$. The edge (C)-(E) is set to ∞ or can be removed because it coincides with a path in List A. Then Dijkstra's algorithm is used to find S_1^2 :

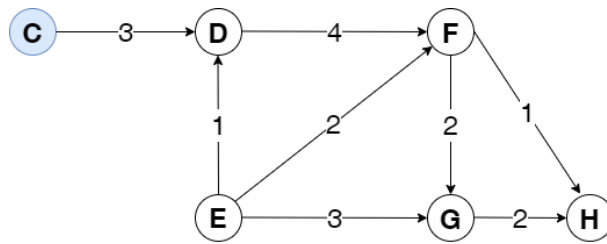


Figure 2: The graph after removing (C)-(E).

S	C	D	E	F	G	H
\emptyset	0	∞	∞	∞	∞	∞
C	0	3	∞	∞	∞	∞
D	0	3	∞	7	∞	∞
F	0	3	∞	7	9	8
H	0	3	∞	7	9	8
G	0	3	∞	7	9	8

We have $S_1^2 = (C) - (D) - (F) - (H)$

$A_1^2 = R_1^2 + S_1^2 = (C) - (D) - (F) - (H)$ with total weight of 8 is added to List B. Then we restore the graph to its original state.

- Now, vertex (E) of A^1 becomes the spur vertex with $R_2^2 = (C) - (E)$, the edge (E)-(F) is removed. Dijkstra's algorithm is used to find S_2^2 :

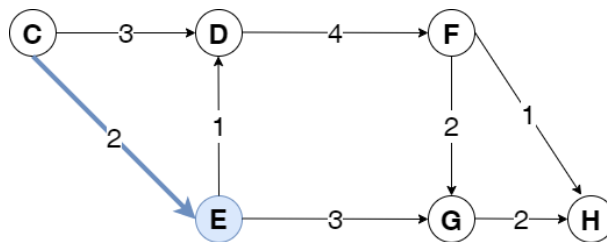


Figure 3: The graph after removing (E)-(F).

S	E	D	F	G	H
\emptyset	0	∞	∞	∞	∞
E	0	1	∞	3	∞
D	0	1	5	3	5
G	0	1	5	3	5
F	0	1	5	3	5

We have $S_2^2 = (E) - (G) - (H)$

$A_2^2 = R_2^2 + S_2^2 = (C) - (E) - (G) - (H)$ with total weight of 7 is added to List B. Then we restore the graph to its original state.

- Vertex (F) of A^1 becomes the spur vertex with $R_3^2 = (C) - (E) - (F)$, the edge (F)-(H) is removed.

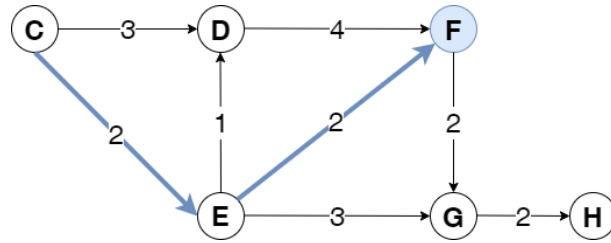


Figure 4: The graph after removing (F)-(H).

Obviously there is only one solution for S_3^2 now, that is (F)-(G)-(H). Therefore, $A_3^2 = (C) - (E) - (F) - (G) - (H)$ with total weight of 8.

- Among the three paths in List B, $A_2^2 = (C) - (E) - (G) - (H)$ is chosen to become A_2^2 because of its lowest weight.
- Finally, we continue working to find the 3rd shortest path, A^3 :
 - Vertex (C) of A^2 is the spur vertex with $R_1^3 = (C)$, the edge (C)-(E) is removed. Since all the other vertices has already been in A^1 and A^2 except D, then we have $A_1^3 = (C) - (D) - (F) - (H)$ with total weight of 8.
 - Vertex (E) of A^2 is the spur vertex with $R_2^3 = (C) - (E)$, the edge (E)-(G) is removed. Since the edge (E)-(F) has already belonged to A^1 , the only one left is (E)-(D), then $A_2^3 = (C) - (E) - (D) - (F) - (H)$ with total weight of 8.
 - Vertex (G) of A^2 is the spur vertex, the edge (G)-(H) is remove. Then there is no other way to reach (H) from (G) so in this case no spur path exists.
 - A_1^3 and A_2^3 have the same weight so we can choose any of them. Here we pick $A_1^3 = (C) - (D) - (F) - (H)$ to be A^3 .

In conclusion, top 3 shortest loop-less path from (C) to (H) are:

$$A^1 = (C) - (E) - (F) - (H), \text{ Total Weight} = 5$$

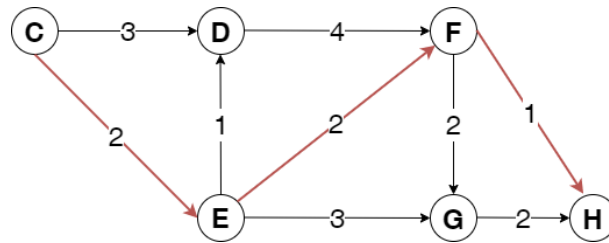


Figure 5: The 1st shortest loop-less path.

$$A^2 = (C) - (E) - (G) - (H), \text{ Total Weight} = 7$$

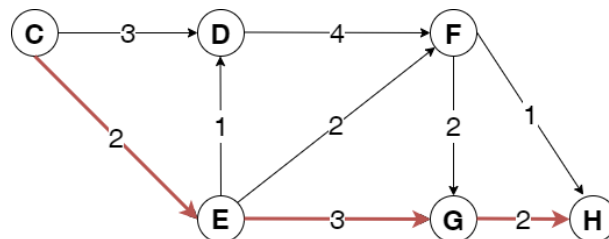


Figure 6: The 2nd shortest loop-less path.

$$A^3 = (C) - (D) - (F) - (H), \text{ Total Weight} = 8$$

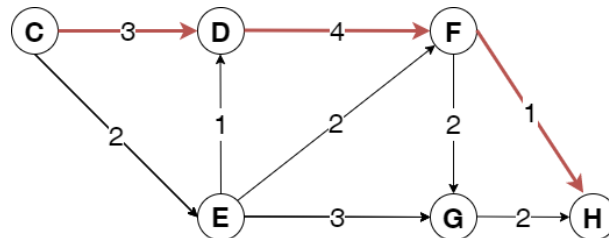


Figure 7: The 3rd shortest loop-less path.

6.2 Example 2:

Find top 2 shortest paths without loops from (s) to (t) in the following graph:

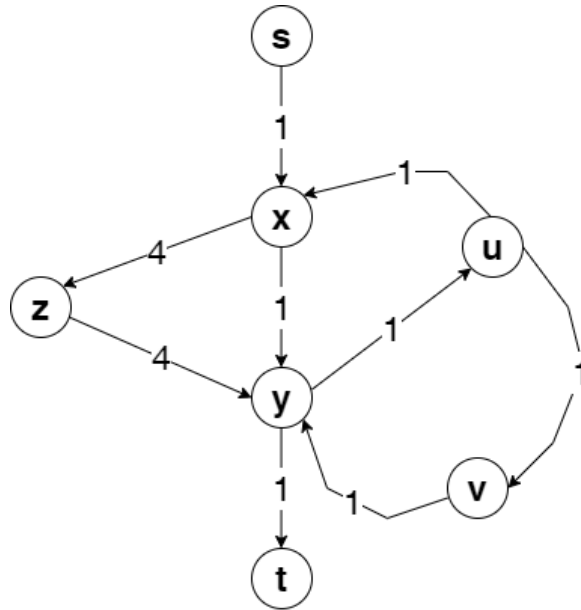


Figure 8: The given graph.

- First of all, using Dijkstra's algorithm helps us the shortest path A^1 in the graph, which is $A^1 = (s) - (x) - (y) - (t)$ with total weight = 3.

S	s	t	u	v	x	y	z
\emptyset	0	∞	∞	∞	∞	∞	∞
s	0	∞	∞	∞	1	∞	∞
x	0	∞	∞	∞	1	2	5
y	0	3	3	∞	1	2	5
u	0	3	3	4	1	2	5
v	0	3	3	4	1	2	5

- Secondly, we going to find A^2 :
 - s is the spur vertex with $R_1^2 = (s)$. Removing the edge (s)-(x) makes no sense because there no exists the spur path S_1^2 from (s) to (t). Thus A_1^2 does not exist too.

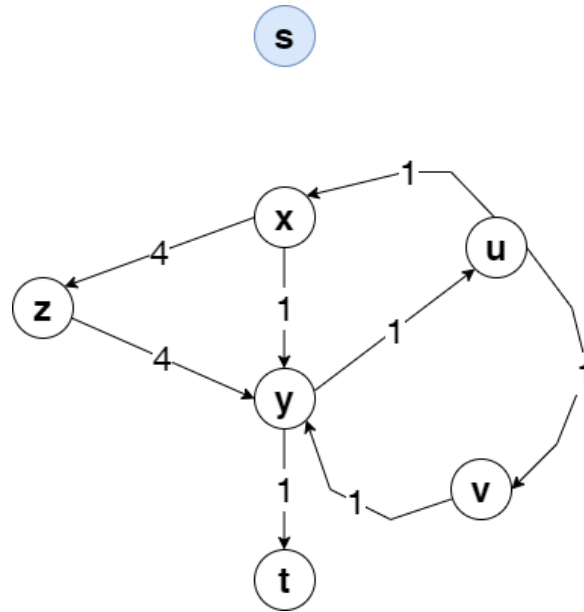


Figure 9: The graph after removing (s)-(x).

- x is the spur vertex with $R_2^2 = (s) - (x)$, removing the edge (x)-(y).

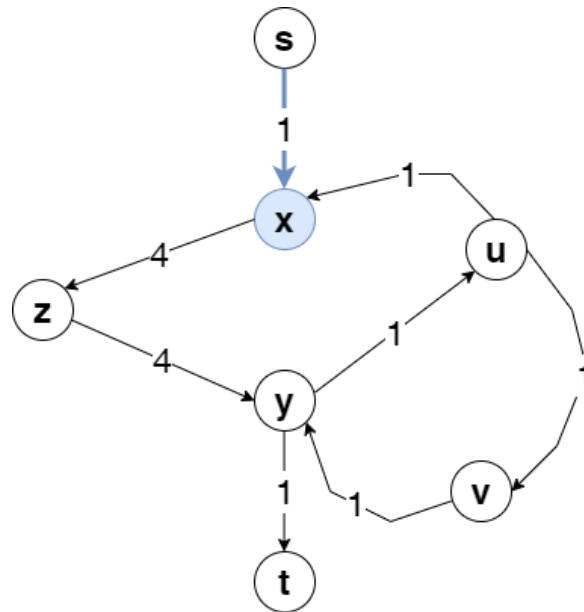


Figure 10: The graph after removing (x)-(y).

Now, we can obtain $S_2^2 = (x) - (z) - (y) - (t)$ since the removal of (x)-(y) makes this the only loop-less way for the spur path.

Therefore $A_2^2 = (s) - (x) - (z) - (y) - (t)$ with total weight = 10.

- Similar to the case s is the spur vertex, when y is the spur vertex, the removal of the edge $(y)-(t)$ is obligated. However, this leads to a dead-end since there is no spur path from (y) to (t) . As a result, A_3^2 does not exist.

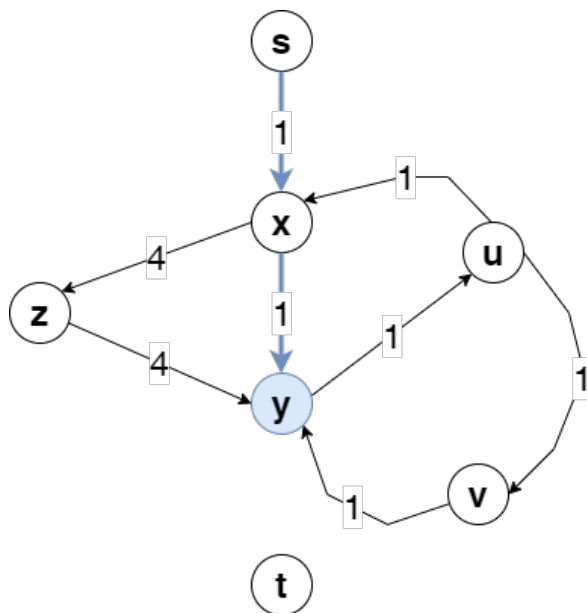


Figure 11: The graph after removing $(y)-(t)$.

- Therefore, we can say that $A^2 = A_2^2 = (s) - (x) - (z) - (y) - (t)$

In conclusion, top 2 shortest paths without loops from (s) to (t) are:

$$A^1 = (s) - (x) - (y) - (t), \text{ Total Weight} = 3$$

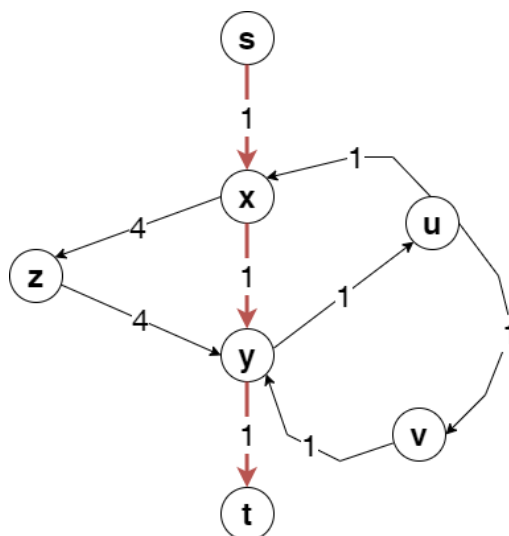


Figure 12: The 1st shortest path.

$$A^2 = (s) - (x) - (z) - (y) - (t), \text{ Total Weight} = 10$$

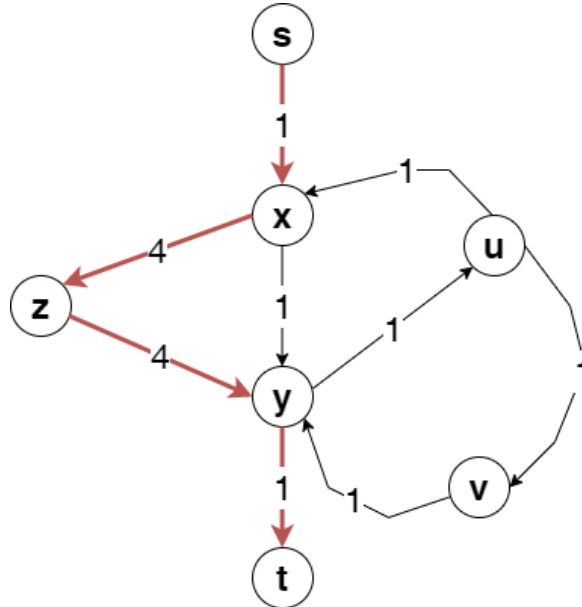


Figure 13: The 2nd shortest path.

6.3 Example 3:

Find top 3 shortest loop-less path from (s) to (t) in the following graph:

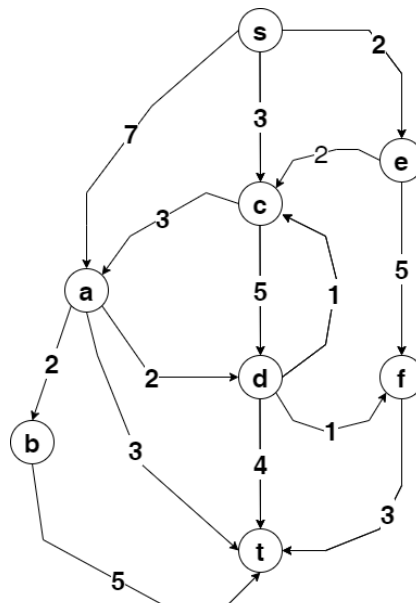


Figure 14: The given graph.

- Initially, we are going to find the shortest loop-less path A^1 in the graph. Using Dijkstra's algorithm give us $A^1 = (s) - (c) - (a) - (t)$ with total weight of 9.

S	s	a	b	c	d	e	f	t
\emptyset	0	∞	∞	∞	∞	∞	∞	∞
s	0	7	∞	3	∞	2	∞	∞
e	0	7	∞	3	∞	2	7	∞
c	0	6	∞	3	8	2	7	∞
a	0	6	8	3	8	2	7	9
f	0	6	8	3	8	2	7	9
d	0	6	8	3	8	2	7	9
b	0	6	8	3	8	2	7	9

- Next, we need to compute the second shortest loop-less path A^2 :
 - s is the spur vertex with $R_1^2 = (s)$, the edge (s)-(c) is removed.

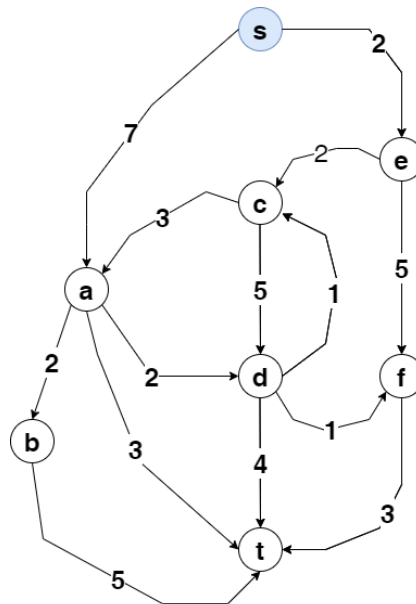


Figure 15: The graph after removing (s)-(c).

S	s	a	b	c	d	e	f	t
\emptyset	0	∞	∞	∞	∞	∞	∞	∞
s	0	7	∞	∞	∞	2	∞	∞
e	0	7	∞	4	∞	2	7	∞
c	0	7	∞	4	9	2	7	∞
a	0	7	9	4	9	2	7	10
f	0	7	9	4	9	2	7	10
d	0	7	9	4	9	2	7	10
b	0	7	9	4	9	2	7	10

Base on the Dijkstra's algorithm, we have $S_1^2 = A_1^2 = (s) - (a) - (t)$ with total weight of 10.

- c is the spur vertex with $R_2^2 = (s) - (c)$, the edge (c)-(a) is removed.

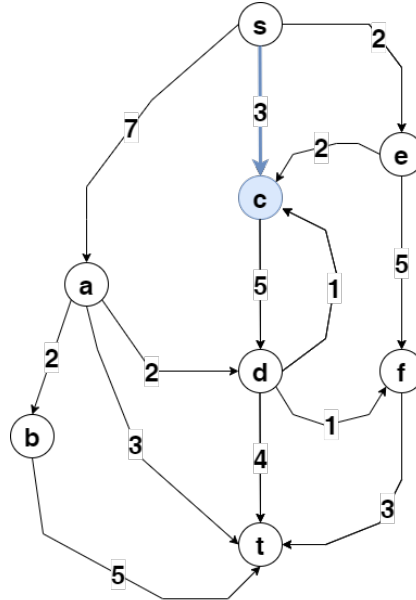


Figure 16: The graph after removing (c)-(a).

S	c	a	b	d	e	f	t
\emptyset	0	∞	∞	∞	∞	∞	∞
c	0	∞	∞	5	∞	∞	∞
d	0	∞	∞	5	∞	6	9
f	0	∞	∞	5	∞	6	9
t	0	∞	∞	5	∞	6	9

According to the Dijkstra's scanning, we can find out that $S_2^2 = (c) - (d) - (t)$.

Thus, $A_2^2 = R_2^2 + S_2^2 = (s) - (c) - (d) - (t)$ with total weight of 12.

- a is the spur vertex with $R_3^2 = (s) - (c) - (a)$, the edge (a)-(t) is removed.

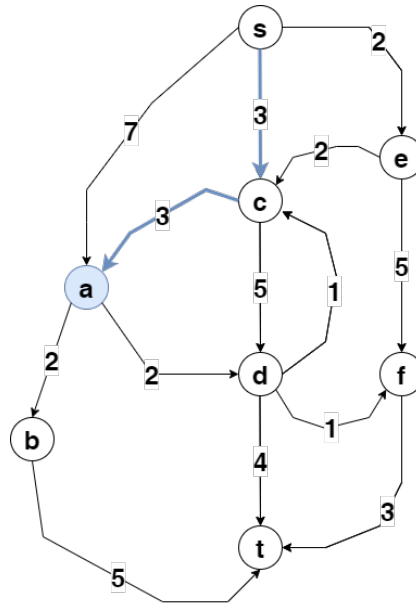


Figure 17: The graph after removing (a)-(t).

S	a	b	d	f	t	e
\emptyset	0	∞	∞	∞	∞	∞
a	0	2	2	∞	∞	∞
b	0	2	2	∞	7	∞
d	0	2	2	3	6	∞
f	0	2	2	3	6	∞
t	0	2	2	3	6	∞

Based on the Dijkstra's algorithm, $S_3^2 = (a) - (d) - (t)$.

Then, we have $A_3^2 = R_3^2 + S_3^2 = (s) - (c) - (a) - (d) - (t)$ with total weight of 12.

- Among the 3 solution for A^2 , A_1^2 is the one with smallest weight so we choose it to become A^2 .
- Finally, we compute the third shortest loop-less path for this graph:
 - $A^2 = (s) - (a) - (t)$. Vertex s of A^2 become the spur vertex with $R_1^3 = (s)$, the edges (s)-(a) and (s)-(c) is removed to make A_1^3 , if it can be found, different from A^2 and A^1 .

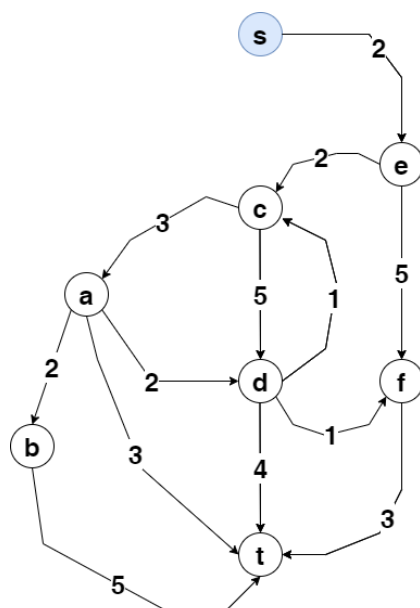


Figure 18: The graph after removing (s)-(c) and (s)-(a).

S	s	a	b	c	d	e	f	t
\emptyset	0	∞	∞	∞	∞	∞	∞	∞
s	0	∞	∞	∞	∞	2	∞	∞
e	0	∞	∞	4	∞	2	7	∞
c	0	7	∞	4	9	2	7	∞
a	0	7	9	4	9	2	7	10
f	0	7	9	4	9	2	7	10
b	0	7	9	4	9	2	7	10
d	0	7	9	4	9	2	7	10

Using the Dijkstra's algorithm gives us $S_1^3 = (s) - (e) - (c) - (a) - (t)$, which is also A_1^3 with total weight of 10.

- When a is the spur vertex with $R_2^3 = (s) - (a)$, the edge (a)-(t) is removed.

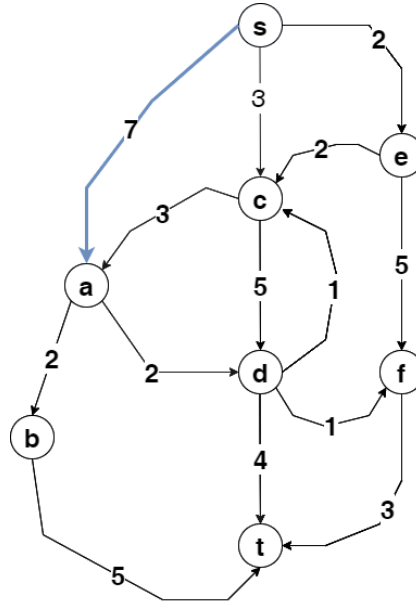


Figure 19: The graph after removing (a)-(t).

S	a	b	c	d	e	f	t
\emptyset	0	∞	∞	∞	∞	∞	∞
a	0	2	∞	2	∞	∞	∞
b	0	2	∞	2	∞	∞	7
d	0	2	3	2	∞	3	6
c	0	2	3	2	∞	3	6
f	0	2	3	2	∞	3	6
t	0	2	3	2	∞	3	6

Using Dijkstra's algorithm, we have $S_2^3 = (a) - (d) - (t)$ with total weight = 6.

Thus, we have $A_2^3 = R_2^3 + S_2^3 = (s) - (a) - (d) - (t)$ with total weight of 13.

- It is clear that A_1^3 has the same total weight with A^2 , in fact it is one of a solution of A_1^2 . However, the usage of Dijkstra's algorithm give a unique solution for A_1^2 because if the new path has exact weight with the previous we don't update it in the scanning table.
- But in this circumstance, we can not choose A_2^3 to be A^3 . If we take a look at $A_2^2 = (s) - (c) - (d) - (t)$ and $A_3^2 = (s) - (c) - (a) - (d) - (t)$, both have 12 in total weight which is smaller than A_2^3 . Since both paths A_2^2 and A_3^2 's weights are equal, we can pick any one of them. Here i choose $A_2^2 = (s) - (c) - (d) - (t)$, total weight = 12 to be A^3 because it contains smaller number of edges in the path.
- This is one of the drawbacks of Yen's algorithm that an improvement to reduce the time spending to find A^3 will be indicated in the last section of this part.

In conclusion top 3 shortest loop-less paths are:

$$A^1 = (s) - (c) - (a) - (t), \text{ total weight} = 9.$$

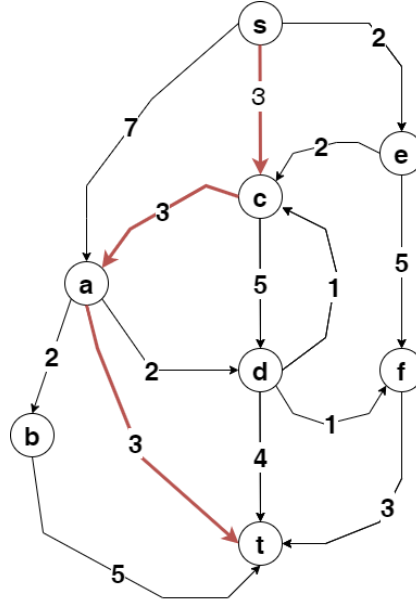


Figure 20: The 1st shortest loop-less path.

$$A^2 = (s) - (a) - (t), \text{ total weight} = 10.$$

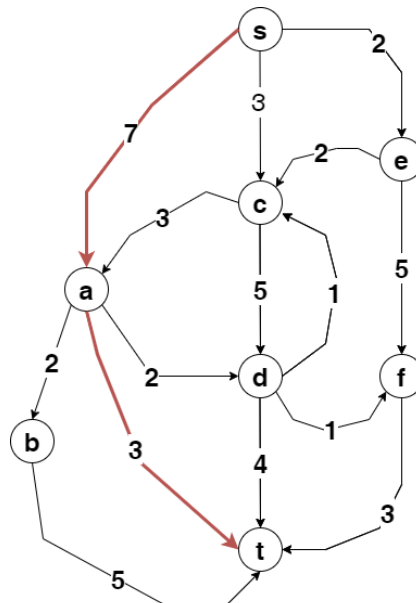


Figure 21: The 2nd shortest loop-less path.

$$A^3 = (s) - (c) - (d) - (t), \text{ total weight} = 12.$$

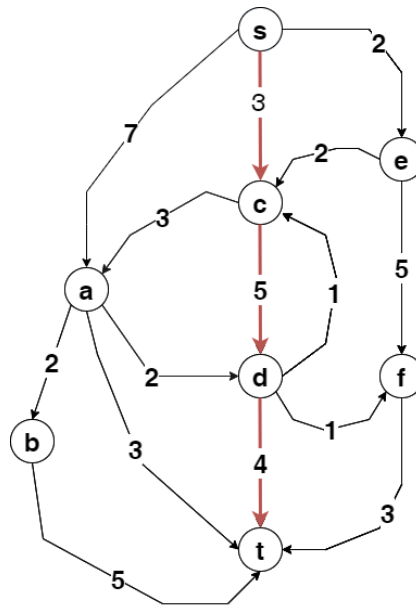


Figure 22: The 3rd shortest loop-less path.

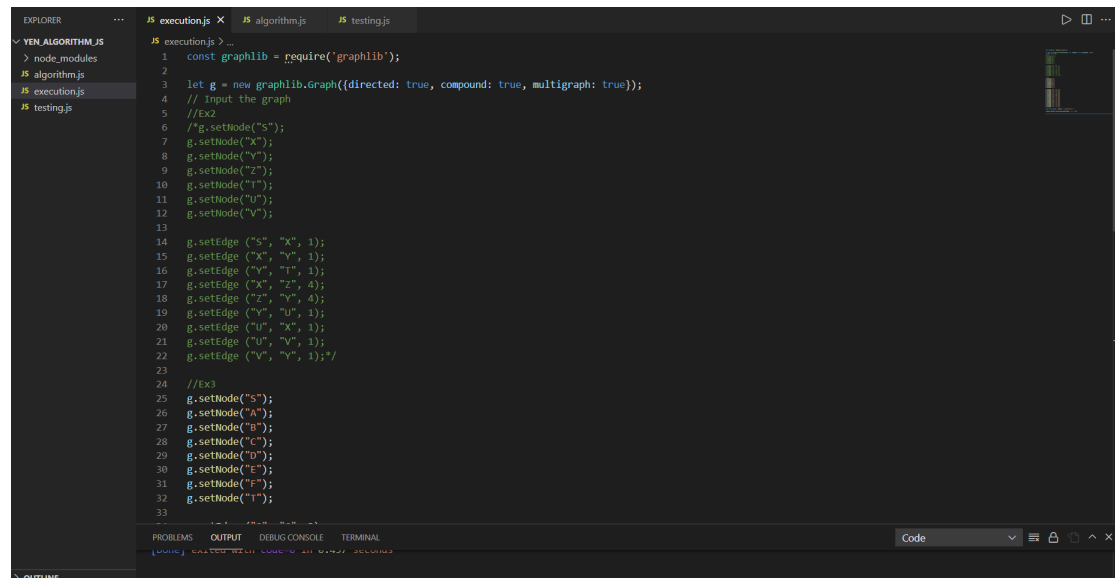
7 Recheck the results with JavaScript Program

The implementation of the program is the combining of my work with some sources in the internet, one of which is from an incognito github user Tomer953. Here is the link: <https://github.com/tomer953/k-shortest-path>

The format of output of the program should include the total weight, weight of each edge in the path and the path traversal. Unfortunately, the program still has some bugs in which we can not view each edge of the path and only the traversal of the first shortest path are visible. About this issue, I'm still trying to figure out the fix but it seems like it can not be done before the deadline.

However, the program still calculates correctly the total weight and shows how many edges containing in the path. This is still useful for us to verify whether our theoretical calculation above is correct or not.

All the source code will be included in the zip file together with the report.



```

1  const graphlib = require('graphlib');
2
3  let g = new graphlib.Graph({directed: true, compound: true, multigraph: true});
4  // Input the graph
5  //Ex2
6  g.setNode("S");
7  g.setNode("X");
8  g.setNode("Y");
9  g.setNode("Z");
10 g.setNode("U");
11 g.setNode("V");
12 g.setNode("W");
13
14 g.setEdge("S", "X", 1);
15 g.setEdge("X", "Y", 1);
16 g.setEdge("Y", "T", 1);
17 g.setEdge("X", "Z", 4);
18 g.setEdge("Z", "Y", 4);
19 g.setEdge("Y", "U", 1);
20 g.setEdge("U", "X", 1);
21 g.setEdge("U", "V", 1);
22 g.setEdge("V", "Y", 1);"/
23
24 //Ex3
25 g.setNode("S");
26 g.setNode("A");
27 g.setNode("B");
28 g.setNode("C");
29 g.setNode("D");
30 g.setNode("E");
31 g.setNode("F");
32 g.setNode("T");
33

```

Figure 23: The outline of the program.

The file execution.js is where we input the graph and run Yen's algorithm to find K-shortest paths. The file algorithm.js contains the implement of Yen's algorithm, while the folder node-modules is a Graphlib library provided by JavaScript gives us data structure for graphs along with many other methods to work with them.

7.1 Example 1:

We input the graph in the file execution.js and run it as in the following figure:

```
1  const graphlib = require('graphlib');
2
3  let g = new graphlib.Graph({directed: true, compound: true, multigraph: true});
4  // Input the graph
5  g.setNode("C");
6  g.setNode("D");
7  g.setNode("E");
8  g.setNode("F");
9  g.setNode("G");
10 g.setNode("H");
11
12 g.setEdge ("C", "D", 3);
13 g.setEdge ("C", "E", 2);
14 g.setEdge ("E", "D", 1);
15 g.setEdge ("E", "F", 2);
16 g.setEdge ("E", "G", 3);
17 g.setEdge ("G", "H", 2);
18 g.setEdge ("D", "F", 4);
19 g.setEdge ("F", "G", 2);
20 g.setEdge ("F", "H", 1);
21
22 const Yen_running = require ('./algorithm.js');
23
24 console.log(Yen_running.Yen_algorithm(g, 'C','H',3));
```

Figure 24: The input graph.

```
[Running] node "c:\Users\acer\Downloads\Yen_algorithm Js\execution.js"
[
  {
    Total_Weight: 5,
    edges: [ [Object], [Object], [Object] ],
    traversal: [ 'C', 'E', 'F', 'H' ]
  },
  {
    Total_Weight: 7,
    edges: [ [Object], [Object], [Object] ],
    traversal: [ 'C' ]
  },
  {
    Total_Weight: 8,
    edges: [ [Object], [Object], [Object] ],
    traversal: [ ]
  }
]

[Done] exited with code=0 in 0.477 seconds
```

Figure 25: The output of the program.

The results match with those obtained by theory:

$A^1 = (C) - (E) - (F) - (H)$ has 3 edges and total weight = 5.

$A^2 = (C) - (E) - (G) - (H)$ has 3 edges and total weight = 7.

$A^3 = (C) - (D) - (F) - (H)$ has 3 edges with total weight = 8.

7.2 Example 2:

```
1  const graphlib = require('graphlib');
2
3  let g = new graphlib.Graph({directed: true, compound: true, multigraph: true});
4  // Input the graph
5  g.setNode("S");
6  g.setNode("X");
7  g.setNode("Y");
8  g.setNode("Z");
9  g.setNode("T");
10 g.setNode("U");
11 g.setNode("V");
12
13 g.setEdge ("S", "X", 1);
14 g.setEdge ("X", "Y", 1);
15 g.setEdge ("Y", "T", 1);
16 g.setEdge ("X", "Z", 4);
17 g.setEdge ("Z", "Y", 4);
18 g.setEdge ("Y", "U", 1);
19 g.setEdge ("U", "X", 1);
20 g.setEdge ("U", "V", 1);
21 g.setEdge ("V", "Y", 1);
22
23 const Yen_running = require ('./algorithm.js');
24
25 console.log(Yen_running.Yen_algorithm(g, 'S','T',2));
```

Figure 26: The input graph.


```
[Running] node "c:\Users\acer\Downloads\Yen_algorithm_Js\execution.js"
[
  {
    Total_Weight: 3,
    edges: [ [Object], [Object], [Object] ],
    traversal: [ 'S', 'X', 'Y', 'T' ]
  },
  {
    Total_Weight: 10,
    edges: [ [Object], [Object], [Object], [Object] ],
    traversal: [ 'S' ]
  }
]

[Done] exited with code=0 in 0.443 seconds
```

Figure 27: The output of the program.

The results match with those obtained by theory:

$A^1 = (s) - (x) - (y) - (t)$ has 3 edges and total weight = 3.

$A^2 = (s) - (x) - (z) - (y) - (t)$ has 4 edges and total weight = 10.

7.3 Example 3:

```
1  const graphlib = require('graphlib');
2
3  let g = new graphlib.Graph({directed: true, compound: true, multigraph: true});
4  // Input the graph
5  g.setNode("S");
6  g.setNode("A");
7  g.setNode("B");
8  g.setNode("C");
9  g.setNode("D");
10 g.setNode("E");
11 g.setNode("F");
12 g.setNode("T");
13
14 g.setEdge("S", "C", 3);
15 g.setEdge("S", "A", 7);
16 g.setEdge("S", "E", 2);
17 g.setEdge("C", "A", 3);
18 g.setEdge("C", "D", 5);
19 g.setEdge("A", "D", 2);
20 g.setEdge("A", "T", 3);
21 g.setEdge("A", "B", 2);
22 g.setEdge("B", "T", 5);
23 g.setEdge("D", "C", 1);
24 g.setEdge("D", "F", 1);
25 g.setEdge("D", "T", 4);
26 g.setEdge("E", "C", 2);
27 g.setEdge("E", "F", 5);
28 g.setEdge("F", "T", 3);
29
30
31 const Yen_running = require('./algorithm.js');
32
33 console.log(Yen_running.Yen_algorithm(g, 'S', 'T', 3));
34
```

Figure 28: The input graph.

```
[Running] node "c:\Users\acer\Downloads\Yen_algorithm_js\execution.js"
[
  {
    Total_weight: 9,
    edges: [ [Object], [Object], [Object] ],
    traversal: [ 'S', 'C', 'A', 'T' ]
  },
  {
    Total_weight: 10,
    edges: [ [Object], [Object], [Object] ],
    traversal: []
  },
  {
    Total_weight: 12,
    edges: [ [Object], [Object], [Object] ],
    traversal: [ 'S' ]
  }
]

[Done] exited with code=0 in 0.457 seconds
```

Figure 29: The output of the program.

The results match with those obtained by theory:

$A^1 = (s) - (c) - (a) - (t)$ has 3 edges and total weight = 9.

$A^2 = (s) - (a) - (t)$ has 2 edges and total weight = 10.

$A^3 = (s) - (c) - (d) - (t)$ has 3 edges and total weight = 12.

8 Improvements for Yen's algorithm

Various improvements to Yen's original algorithm have been proposed over the years. The most remarkable one is the use of a heap to store Yen's List B, by that producing an improvement on the performance, however the complexity does not change. A more advanced enhancement is checking for non-existent spur paths (i.e. where the root path exists, but all spur paths from that root have been used previously).

If List B contains enough shortest paths all of the same next minimum length, to produce all the required paths, it is not necessary to extract each path in turn and perform the above calculations; only to extract the required number of paths and place them directly in List A, as no other shortest paths will be found.

Lawler's modification



In 1972, Eugene Lawler presented a modification to Yen's algorithm, such that instead of calculating and then discarding any duplicate paths, they are simply never calculated.

The extra paths occur due to the calculation of spur paths from nodes in the root of the K-shortest paths. Lawler states that it is only necessary to calculate new paths from vertices that were on the spur of the previous shortest path. Consider a vertex on the root of a path, Yen calculates a spur path from this node every time a new K-shortest path is required.

Therefore, it is necessary to keep track of the vertex where each path branched from its root. This vertex marks the point from where the calculation of spur paths starts. When finding more than two shortest paths, the next shortest path will on average branch from the middle of the path and so approximately a 50% improvement in speed is achieved over Yen.



References

- [1] Jin Y.Yen. “Finding the K Shortest Loopless Paths in a Network”. In: *Management Science* 17.11 (1971), pp. 712–716.
- [2] Eugene L.Lawler. “A Procedure for Computing the K Best Solution to Discrete Optimization Problems and its Application to the Shortest Problem”. In: *Management Science* 18.7 (1972), pp. 401–405.
- [3] A.W.Brander and M.C.Sinclair. “A Comparative Study of k-Shortest Path Algorithms”. In: *Performance Engineering of Computer and Telecommunications Systems* (1996). DOI: https://doi.org/10.1007/978-1-4471-1007-1_25.
- [4] Grégoire Scano, Marie-José Huguet, and Sandra Ulrich Ngueveu. “Adaptation of k-Shortest Path Algorithms for Transportation Networks”. In: *International Conference on Industrial Engineering and Systems Management* (Oct,2015).

[1, 2, 4, 3]