The k-Shortest Simple Path Problem Using Hub-Labelling

Eleni Batziou

April 25, 2018

1 Introduction

In the present work, we introduce an efficient algorithm for the *k-Shortest Simple Paths* ($k - SSP$) problem. By this definition, one refers to the k shortest paths from a source s to a sink t in a graph $G = (V, E)$, without including cycles. Cycles are defined as multiple visits in a vertex or edge inside a path.
The core ingredient in our algorithm is the use of a landmark labelling technique. This method, called *Hub-Labelling*, basically constitutes a method of storing and efficiently extracting all-pairs shortest path distances in a graph.
The basis of most current methods on the $k - SSP$ problem is Yen's algorithm. This algorithm uses the notion of detour edges, namely gradually includes non-shortest path edges as small detours in an already calculated shortest path. Yen's algorithm works by dividing a shortest path in a prefix and suffix path, for each node in this path uses Dijkstra to calculate the shortest path from the deviating node to the sink.
Our algorithm relies on Yen's technique, and provides an alternative method of calculating the shortest simple path from the deviating node to the sink. We use hub-labelling to improve upon Dijkstra's bound, and output the shortest simple path in a more efficient manner, rather than exhaustively examining the entire graph for every deviating node.

2 Yen's Algorithm for the $k - SSP$ Problem

Yen's algorithm, published by Jin Y. Yen in 1971, computes k single-source shortest simple (loopless) paths in a graph with non-negative edge costs. Initially, the algorithm uses some known algorithm to compute the initial shortest simple path from a source s to a sink t, and then proceeds by finding $k - 1$ deviations of the optimal path.
Formally, the algorithm receives as **input** a graph $G = (V, E)$, a source s, a sink t, and the number K, which is the number of shortest paths required from s to t. The **output** produces by the algorithm are the K-SSPs, stored inside a container A, in decreasing order from $A[1]$ to $A[k]$.
The process is divided in two main parts. The first part consists of calculating the shortest simple path from source s to sink t. The second part consists of finding $k - 1$ deviations of this particular path.
The algorithm makes use of two containers: A for the actual SSPs, and B for the potential SSPs. Therefore, A^1 refers to the smaller SSP from s to t. One can deploy any shortest path algorithm to determine this path. In Yen's original version, Dijkstra's algorithm is used to calculate A^1.
Our goal is to determine the paths A^i, where $2 \leq i \leq k$. In order to calculate A^k, the k-th SSP, we assume that A^{k-1} has previously been found. First, one has to determine all the deviations, for each node in the $(k - 1)$-th SSP, and then to chose the best solution among those candidate paths. The number of deviating paths from a single path is equal to the number of nodes in this A^{k-1} path.
The last phase can be described by the following procedure: Let i be the deviating node, or $spurNode$ as it is mentioned in the original version, when trying to determine A^k. First, we find the prefix path (namely, the subpath in A^{k-1} including all nodes before i in the previous SSP). This constitutes the $rootPath$. This means that, for each iteration of j from 1 to $k - 1$, we remove the first i nodes of the $(j - 1)$-th SSP, in order to result with a unique SSP, not cycling with nodes from the previous SSPs. Additionally, we remove the edge between $e = (i, i + 1)$ that was included in the previous $(j - 1)$-th SSP.
Then, we remove the nodes and edges of the $rootPath$ of $j - 1$, plus the edge e. We then run a shortest path algorithm, from the $spurNode$ i to sink t, in order to result with the remaining SSP, the $replacementPath$. The removal of the previous edges ensures that the new path is different from the $(j - 1)$-th SSP.
Finally, from the shortest path property, by combining the $rootPath$ and the $replacementPath$, we result with ${A^k}_i$, that is the replacement path for each deviating node i from path $A[k - 1]$. The latter path is then added to the container B, while restoring all the original nodes and edges in the graph G.
After calculating a replacement path for all deviating nodes from $A[k - 1]$, we extract the path A^k from B as the path with the lowest cost. The path is inserted to A, and the algorithm continues with the next iteration. If the number of paths that are present inside B is equal or greater to the number of paths remaining to be found, then the algorithm terminates by moving these shortest paths from B to A.
Our contribution to Yen's algorithm, described in detail in **Section 4**, is that we present an alternative method of computing the replacement path problem. In our technique, we use hub-labelling, in addition to a modified Breadth-First Search, in order to determine the shortest path from $spurNode$ to sink t.

Algorithm 1 Modified Yen's Algorithm

```
function modifiedYens(Shortest path list A, Labels L, Graph G, source s, sink t, spurNode x, int K)
    // Determine the shortest path from the source s to the sink t.
    A[0] = Dijkstra(G, s, t);
    // Initialize the heap to store the potential kth shortest path.
    B = [];
    // Here we maintain information about a potential change in level for the nodes of the graph. Change of level in the BFS tree means that the shortest path information on the label may not be usable, as it may include nodes from a previous SSP.
    modified = [];
    for k from 1 to K do
        // The spur node ranges from the first node to the next to last node in the previous k-shortest path.
        for i from 0 to size(A[k − 1]) − 1 do
            // Spur node is retrieved from the previous k-shortest path, k-1.
            spurNode = A[k-1].node(i);
            // The sequence of nodes from the source to the spur node of the previous k-shortest path.
            rootPath = A[k-1].nodes(0, i);
            // Run a BFS in the original graph, and output the level of each node.
            oldLevel, backEdgeCount = levelBFS(G, spurNode);
            for each path p in A do
                if rootPath == p.nodes(0, i) then
                    // Remove the links that are part of the previous shortest paths which share the same root path.
                    remove p.edge(i,i + 1) from G;
                end if
            end for
            for each node rootPathNode in rootPath except spurNode do
                remove rootPathNode from G;
            end for
            // Run a BFS in the modified graph, excluding nodes from previously found SSPs that may cause cycling.
            newLevel, backEdgeCount = levelBFS(G, spurNode);
            for each node u in G do
                if oldLevel[u] != newLevel[u] then modified[u] = 1
                end if
            end for
            // Calculate the spur path from the spur node to the sink.
            spurPath = replacementPath(G, L, spurNode, sink, modified, backEdges);
            // Entire path is made up of the root path and spur path.
            totalPath = rootPath + spurPath;
            // Add the potential k-shortest path to the heap.
            B.append(totalPath);
            // Add back the edges and nodes that were removed from the graph.
            restore edges to G;
            restore nodes in rootPath to G;
        end for
    end for
end function
```

3 Akiba's Algorithm for Pruned Landmark Labelling

Akiba et al. proposed an efficient exact method for shortest-path distance queries in large networks. This method uses preprocessing in a graph G, in order to result in a container with all-pairs shortest paths. It belongs to a broader family of algorithms performing *distance labelling*. In this setting, for each node $u \in V$, we store a subset $S(u) \subseteq V$ of other nodes, such that, for each queried pair $u, v \in V$ we get:

$$d'(u,v) = min_{w \in S(u) \cap S(v)} d(u,w) + d(w,v)$$

where d represents the shortest path distance between two nodes.
A core notion in this study is that of a *hubset*. A *hubset* $H_u \subseteq V$ of a node u in a graph G is a set of other nodes such that, if P_{uv} is the shortest path from u to v in G:

$$\forall u, v \ \exists a \in H_u \cap H_v, \ a \in P_{uv}$$

This algorithm takes as input a graph $G = (V, E)$ and an ordering of the vertices, and outputs the hub-set of each node, with a few modifications.
The output of the algorithm complies to the following form:

$$L(u) = \{(w, d(u,w))\}_{w \in H_u}$$

where $L[u]$ is the *label* of u. The shortest path distance $d(u,v)$ between two vertices u and v can therefore be computed as $min\{\delta + \delta' \mid (w,\delta) \in L(u), \ (w,\delta') \in L(v)\}$. The family of labels $\{L(u)\}$ is called a *2-hop cover*.
The main modification applied in this paper is the introduction of the notion of *pruning*. A naive implementation of the above algorithm for labelling yields a complexity of $O(nm)$ preprocessing time, as well as $O(n^2)$ space for storage of information. The pruning process takes place during the BFS searches.
We assume that S is a set of vertices and suppose that we already have labels capable of producing the correct distance between any two vertices, if a shortest path between them passes through a vertex in S. In this case, if there exists a vertex $w \in S$ such that $d(v,u) = d(v,w) + d(w,u)$, we prune u. This means that the resulting label $L(v)$ does not contain information about u, but only about w, since the shortest path from v to u passes from an already calculated vertex w.
The result of the above process is an improved, reduced label size. This is a crucial improvement, since for the majority of hub-labelling procedures and algorithms, the label size is not known or can be arbitrarily large. There is not an exact known upper bound for the label size in any algorithm, but we do know that, by using this pruned landmark labelling technique, the label size is significantly smaller.
To conclude, Akiba's algorithm provides a good trade-off between label size and query time. An important aspect of this algorithm is that it produces label with minimum size, and their minimality can be formally proved. Thus, by employing this algorithm to produce a labelling in a graph, we can guarantee that our complexity remains minimal. By query time, one defines the time required to output the exact shortest path between any two vertices in a graph.

4 Replacement Path Algorithm

In the current section, we present our version for the replacement path problem, using information extracted by the pruned hub-labelling procedure.

Our algorithm takes as *input* a graph G, $spurNode\ s$, sink t, the container A of the previously calculated SSPs, and two variables, a matrix $modified$ and a flag $backEdges$. These variables result from the modified BFS algorithm. The *output* of the algorithm is the $spurPath$, the SSP from $spurNode$ to t. As discussed above, a key feature of our algorithm is the modified BFS process, $levelBFS$, described in **Section 5**. This procedure enables us to check if some information from the labels results in cycling with previous SSPs or not.

Instead of running a Dijkstra, as is the initial version of Yen's algorithm, we propose a different version, using information from labels, created by a hub-labelling process. Here, we make use of Akiba's algorithm for pruned landmark labeling, which provides an optimal tradeoff between query time and label size. Label size is crucial, since it defines the complexity of our algorithm.

In most hub labeling algorithms, the maximum size of a label is not known or upper bounded. Therefore, even though we can achieve a good query time, such as $O(2L)$, where L denotes the maximum size of a label, we do not know how much information needs to be stored and examined, and the preprocessing may result in a higher complexity than previously known algorithms. Akiba's algorithm however, by using pruning, results in a much smaller complexity, namely it computes a labelling in $O(n(n+m))$, with label size upper bounded by $O(n)$ and query time $O(n)$. The upper bound on the total size of *all* the labels for the n nodes of the graph remains to be proved..

Our algorithm progresses by following the information extracted from the labels. Starting from the spurNode, we proceed by examining its label, and adding nodes in the queue, if we are allowed to do so. The latter means that, if this label contains a node whose path is not simple, we do not enqueue this node. This check is performed by examining if this target node is modified or not, by the comparison of $oldLevel$ and $newLevel$, as discussed below. If it is, the node is not enqueued. If not, the node is enqueued, and we examine its label, by following the exact same process, when it is dequeued from the queue.

The only case where we actually take into account the data from one node i's label towards a modified node, is when i is adjacent to the modified node. Only then can we be sure that there is no cycling in the resulting path with nodes used in previous SSPs, since the two nodes are direct neighbors, therefore there is no cycle possibly occuring.

Another case when we do not enqueue a node is if it is below t in the $newBFS$ tree, and the flag $backEdges$ is set to 0. We've previously set the variable $backEdges$ to represent if there is some node below t in the BFS tree leading to a node in a level before t in the tree. This means that, if $backEdges$ is set to 0, there is no possible path from nodes below t leading up to t. Therefore, the search can stop. Otherwise, for some node j with $newLevel[j] > newLevel[t]$, if $backEdges = 1$ there exists an edge (j, i), with $newLevel[i] <= newLevel[t]$. Therefore, one has to examine the whole graph, even edges below t's level.

An aspect of our algorithm that is worth mentioning is that, if for some node i we know the shortest path to t and t is not modified, then we do not examine or enqueue any of i's neighbors. This statement holds, since the labels contain the shortest path distance, therefore if t is not modified, that means that we already know the SSP from i to t. This check aids in the optimization of our algorithm, and is a key ingredient of its improved performance comparing to Dijkstra's algorithm. The only remaining aspect to be improved is the path from i to t, given that we need to find the exact nodes to be added in the resulting $spurPath$.

Algorithm 2 Replacement Path calculation using Hub-Labelling

```
function REPLACEMENTPATH(Graph G, Labels L, rootnode root, sink t, modified, backEdges)
    // Output: Shortest path from root to sink t in spurPath
    Q ← empty;
    d = [];
    parent = [];
    tempRoot = root;
    Q.push(tempRoot);
    while Q not empty do
        while (examined[tempRoot]) or ((not backEdges) and (newLevel[tempRoot] > newLevel[t])) do
            tempRoot = Q.pop();
        end while
        if (not modified[t]) and (t in L[tempRoot]) then
            if d[t] ≤ w+d[tempRoot] then
                d[t] = w+d[tempRoot];
                parent[t] = tempRoot;
            end if
        else
            for each node u in L[tempRoot] do
                examined[u] = 1;
                if (not modified[u]) and (not backEdges) and (newLevel[u] > newLevel[t]) then
                    continue; // Go to next iteration of For-loop
                else if (not modified[u]) or ((modified[u]) and (u == neighbour(tempRoot)) then
                    if not examined[u] then
                        Q.push(u);
                    end if
                    if d[u] ≤ w+d[tempRoot] then
                        d[u] = w+d[tempRoot];
                        parent[u] = tempRoot;
                    end if
                end if
            end for
        end if
    end while
    visiting = t;
    reversePath ← empty;
    neighboringQueue ← empty;
    // We trace back the SSP leading from spurNode to sink.
    while visiting != root do
        reversePath.push(visiting);
        visiting = parent[visiting];
    end while
    spurPath = [];
    i ← reversePath.size()-1;
    while i ≥ 0 do
        spurPath[i] = reversePath.pop();
    end while
    return spurPath
end function
```

5 BFS Search and Level Calculation

In this section, we present the procedure used to calculate the *levels* in a BFS tree, namely the unweighted distance, in number of hops, of a node from the root of the BFS tree.
As *input*, $levelBFS$ receives a graph $G_1 = (V, E)$, and *outputs* a matrix representing each node's level in the BFS tree, as well as a flag, $backEdges$, indicating if there exists any path from nodes below t's level in the BFS tree leading back to t.
The purpose of this procedure is the following: If some node has a different level in the BFS tree before the removal of the edges of the previous SSPs, then it is possible that a path leading from another node to it uses some of the edges from the previous SSPs. This can be more clearly illustrated in the example below.

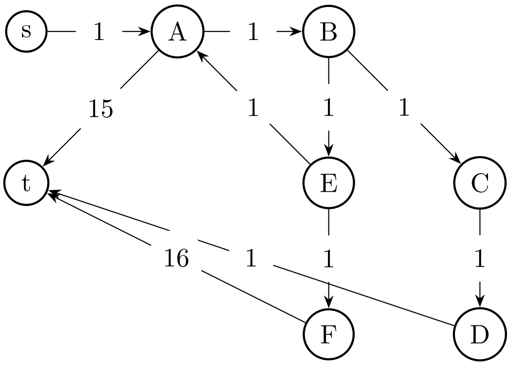

Figure 1: Graph G_1.

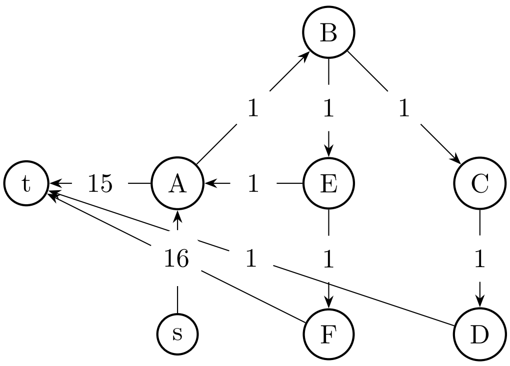

Figure 2: $preBFS$: BFS in the non-modified graph G_1.

In the above graph G_1, we consider s as the source, t as the sink and B as the $spurNode$. Therefore, we compare the outcomes of the two BFS trees rooted at B.
We assume that we have some information concerning the shortest simple path from s to t. This path uses the edges $(s, A), (A, B), (B, C), (C, D), (D, t)$.

Now, we chose B as the $spurNode$, from which we search for a shortest path to t. Then, for the first BFS in the non-modified input graph G_1 rooted at B (the spurNode), A, E, C have oldLevel set to 1, and t, S, F, D, set to 2.
In this graph, by chosing B as the $spurNode$ we result in a $spurPath:\ (s, A), (A, B)$.
Following Yen's technique, we remove the nodes and edges from all shortest paths **before** the spurNode B, that is the nodes of the prefix path (or $rootpath$), and the edges from B that have also been used in previous SSPs. Therefore, we remove nodes s, A from the graph (along with their edges), that create the $rootPath$, as well as edge (B, C).
The $newBFS$ tree for the resulting graph G' is illustrated just below.

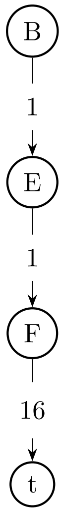

Figure 3: $newBFS$: BFS in the modified graph G_1'.

The levels in the $newBFS$ tree are as follows: $E = 1$, $F = 2$, t= 3. Therefore, $oldLevel[t]! = newLevel[t]$ and hence $modified[t] = 1$. Also, we do not add vertices C, D, since removing edge (B, C), as instructed by Yen's technique, there is no alternative path from B to C or D in the graph, and therefore are unreachable.
After the pruned hub-labelling process, we result with label $L[u]$ for each node u, that contains information on the shortest paths. Our algorithm is not order-sensitive, and therefore the order by which the labelling process is performed is of no significance.
The data is in the form:
$L[u] = \{(v, w)|v :$ destination node of path from u, w: weight of shortest path from u to $v\}$.
Because of the definition of the labels, it is possible that for node E we have: $L[E] = \{(A, 1), (E, 0), (t, 16)\}$, if one assumes that the labelling process covers F before t -this has to do with the order by which we cover the vertices in the pruned BFS.
The reason for the above is that the shortest path from E to t passes by node A, and has a weight of 16.

Since edge (B, C) was removed, the only way of reaching t from B is by passing through E. Therefore, edge (B, E) is bound to exist on the $spurPath$. By using this information from the label, one would result with the following $spurPath$: $(B, E), (E, A), (A, t)$. Following Yen's technique, after chosing the $spurPath$, we concatenate it with the $rootPath$. Therefore, we get the candidate SSP as: $(s, A), (A, B), (B, E), (E, A), (A, t)$, which contains a cycle! This means that we can not use E's shortest path to t, as given from E's label $L[E]$, since its' level has been modified.
As a result, when running the $replacementPath$ procedure to find the shortest path from B to t in the modified graph, excluding previous shortest path edges, we ignore t when we come across it in some node's label. The only case when we can take t into account, is when examining the labels of his neighbours (adjacent vertices).
Concluding, by checking the level of each node, one can verify if the labelling information is actually usable or not, during the $replacementPath$ process.

6 Back Edges in BFS tree

In addition to the BFS procedure described in the previous Section, we add a check for back edges. Here, *backEdges* is a binary flag, demonstrating if there are edges below t in the BFS tree that may lead to t after some number of hops. That means that, if all nodes leading to t are situated above t in the BFS tree, then there is no reason to proceed the search after the level of t. As a result, if there are no back edges leading to upper levels in the **new** BFS tree, when we come across a node below t, we do not take it into account.
However, having back edges to levels above, means that there exists a path from nodes below t's level leading up, above t's level. Therefore, we take into account all edges of the graph in this case.

This argument can be clearly illustrated by the following example.

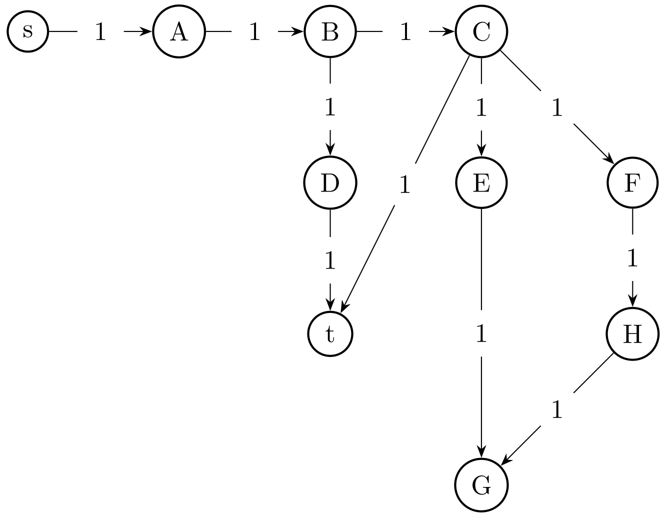

Figure 4: Graph G_2.

For the above graph, we use the same notation for source and sink nodes. The *spurNode* in this case is chosen as node ?. As one can easily observe, there is no path from nodes after E to sink t.
Following Yen's technique, we remove nodes s, A, as they create the *rootPath*, and their respective edges. Furthermore, we remove edge $(?, D)$, as it is the first node of the previous *spurPath*.
The outcome of *levelBFS* for the modified graph is the following.

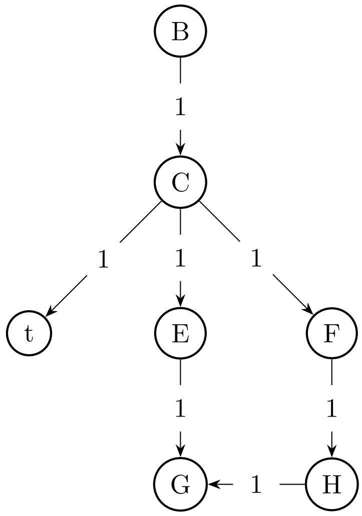

Figure 5: $newBFS$ for modified Graph G_2'.

As one can easily observe in Figures 4 and 5, there is no path from edges E, F, G, H leading to t. Therefore, examining them during the search of our algorithm would only result in additional computational complexity and cost, without yielding any result. In this case, flag $backEdges$ remains 0.
During $levelBFS$, we examine nodes per level. When arriving to nodes equal to t's level, and assign them as temporary roots ($tempRoot$ in our algorithm),we examine if they have any outcoming edges to nodes with a level smaller than t, that is above t in the BFS tree. If yes, then $backEdges$ is set to 1. This means that there is a path from nodes below t's level leading back to t. Otherwise, we continue the search until the whole graph is explored. If there does not exist any node below t with edges above or at the same level as t, then $backEdges = 0$. Therefore, there is no path that may lead to t, when progressing to lower levels.

Algorithm 3 BFS with Level and Back Edge Calculation

```
function levelBFS(Graph G, rootnode root, sink t)
    // Output: level: representing each node's level in the BFS tree, backEdges: if there exists a path from nodes below sink's level that may lead to sink t.
    Q ← empty;
    level[root] = 0;
    marked[root] = 1;
    Q.push(root);
    // If, after the level of sink t there exist any back edges leading to upper levels, that means that we have to explore other levels as well. Otherwise, if beyond t's level, there is no path leading to t, then we stop the exploration at a smaller level.
    backEdges = 0;
    while Q not empty do
        tempRoot = Q.pop();
        for each u in neighbour(tempRoot) do
            if not marked[u] then
                Q.push(u);
                level[u] = level[tempRoot]+1;
                marked[u] = 1;
            else if (not backEdges) and (marked[u]) and (level[tempRoot] ≥ level[t]) and (level[u] ≤ level[t]) then
                backEdges =1;
            end if
        end for
    end while
    return level, backEdges;
end function
```

7 Proof of correctness for the *replacementPath* algorithm

In this proof of correctness, we will prove that our algorithm *replacementPath* correctly determines the simple shortest path distance from *spurNode* to sink t, output in $d[t]$. If this statement holds, then by retracing the parent matrix, one can retrieve the actual shortest path.
Let us assume some indexing on the n nodes of the graph $G = (V, E)$. Now, let s have an index of 0, and we can suppose, without loss of generality, that the indexes are given according to the levels of the newBFS tree. This fact however is rather irrelevant, since the proof stands for any other random indexing.
We will prove our argument by mathematical induction on k, where k is the index of some node in G.
Since distances are initialised at 0, for $k = s = 0$, $d[s] = 0$. Therefore, for the **base case**, the property holds.
We assume that the property holds for every node i with $0 \leq i \leq k$. This means that, for every i between 0 and k, the algorithm computes the correct SSP distance from *spurNode* to i in $d[i]$.
For the last step, we need to prove that the property holds for some node $k+1$. In order to examine node $k+1$, the algorithm must eventually reach it. The only way to reach a node, is by extracting it from the priority queue Q. Nodes are inserted in the priority queue when they are discovered inside a previously examined node's label. Therefore, in order to insert node $k+1$ in the priority queue Q, it must have been in some node's label, say node k's label $L[k]$. This means that: $L[k] = \{(k+1, w(k, k+1)\}$, among others, where $w(k, k+1)$ is the weight of the shortest path, *we will explain why the SP is simple shortly*, from k to $k+1$, as set during the hub labelling procedure.
If just one single node k has it $k+1$ on its label, then this means that $k+1$ has only one incoming edge, and therefore one single SSP.
If more than one nodes have k on their labels, we examine all the cases identically, since the procedure is always the same, resulting from the for-loop. In the following proof, we show that, for any node containing $k+1$ in its label, the algorithm produces the shortest path from that node k to $k+1$. In the case where more than one nodes contain $k+1$, then, as the algorithm covers the graph, it compares the shortest paths produced by possible paths to $k+1$, and eventually keeps the best solution. This check is performed in steps $19-21$ of *replacementPath* algorithm. Since we examine the whole graph, we end up with the correct SSP distance.
In steps $13-17$, we perform an initial check. If the examined node used as temporary root, say node k as stated above, has t on its label and t's level is not modified, therefore there is no cycle possibly occuring, there is no need to examine k's neighbors.
We can prove this fact by contradiction. Assume that one of k's neighbors, say u, has a shorter path from u to t than the path from k to t. In this case, the label of k would be wrong, since it contains the shortest path from k to t, and that path is simple since t is not modified. Therefore, there is no neighbor of k that may lead to a shorter simple path to t.
Now, moving forward to the examination of the SSP towards $k+1$. We assume that $k+1$ is contained in one node's label. As stated above, if $k+1$ exists in

more than one labels, the different candidate SSP distances from different nodes are compared, and the minimum one is chosen.
There are two cases for $k+1$. Either it is a non-modified node, meaning that it is situated in the same level in the $preBFS$ and $newBFS$ trees, either it is modified and $k = neighbor(k+1)$.

- **Case 1** *Same level in the BFS tree*
  This means that no edges from the previous shortest paths are used in the path from $k+1$ to k. Therefore, if we already know the shortest simple path from s to k, then by adding just the weight of the shortest path from k to $k+1$, calculated by the label, we get the shortest path from s to k, by the property of the shortest paths.
  In this case, we may have shortest paths in the labels directly in more than one hops. The former means that k and $k+1$ are not direct neighbors, but the shortest path is guaranteed to be given by the label of $k+1$. Additionally, it is simple, since there is no way that the path can be include nodes from previous SSPs, as they do not affect $k+1$'s level in the tree.

- **Case 2** *Different level in the BFS tree*
  When k is located at a different level in the newBFS tree, that means that there might be some link to the previous shortest paths, that leads to k. So, there may be some weight inside one node?s label that passes from a node that has been already used. In order to avoid that, we do not take into account modified nodes, in the path from s to k.
  Therefore, if we arrive to t from some node k, given that $k+1$ is in k's label, and $modified[k+1] = 1$, this means that k and $k+1$ are adjacent/neighbors. In this case, the path between k and $k+1$ is bound to be simple, since it only consists of one edge. Given that, in the inductive hypothesis, we assumed that for all nodes up to k, we already know that $d[k]$ is the SSP distance from $spurNode$, and in order to arrive to $k+1$ it must exist in some node k's label, then $d[k+1] = d[k] + w(k, k+1)$, which is the exact SSP from $spurNode$ to $k+1$.

Additionally, for steps $21 - 22$, we can prove that the check is correct. This means that, we do not consider nodes below t's level, when there do no exist $backEdges$. This fact is straight-forward, and is presented again in **Section 5**. If $backEdges = 0$, this means that below t's level in the newBFS tree, there is no path leading back to t. Therefore, all nodes below t's level, need not be considered as potential roots in the search process, as their processing may only result in additional computation time. As a result, if this condition holds, we do not examine that node.