Algorithms For Graph Structures

Martin Janda

June 5, 2020

Prague College

Algorithms and Data Structures (FT)

Dominik Pantůček

Shortest Route(s)

ICA2

1971 Words

Semester 2001

Contents

1 Scenario **1**

2 Aim **1**

3 Prerequisite Structures and Functions **1**

3.1 Data Structures 1

3.1.1 Red-Black Trees 1

3.1.2 Graph 2

3.2 Adding Data 2

4 Dijkstra’s Algorithm Without Edge Weights **3**

4.1 Breadth First Search Algorithm 3

4.2 Trace Back Function for Dijkstra’s Algorithm 5

4.2.1 Picking the Neighbor With The Lowest Distance To Finish 7

4.3 Dijkstra Function 7

List of Figures

1 Flowchart for “breadth-first-search!” Function 4

2 Flowchart for “dijksta-trace-back” Function 6

3 Flowchart for “dijkstra!” Function 8

1 Scenario

The assignment provides a set of vertices (cities or road crossings in the E.U.) and a set of edges (major E.U. roads) as commands meant to be passed to an application program interface designed by the student. The student was given two cities and asked to code an algorithm in Clojure that was capable of finding the shortest path (or proving one does not exist) between each city and Prague. The student was given "Nuremberg" and "Palmero".

2 Aim

The aim of the assignment was to describe the implementation of the algorithms specified by the assignment brief. This technical report also adresses the graph and helper structures that have been implemented as they are a large part of the code being submitted. All code in this doccument was written by the author in Clojure a list proccessing programming language.

3 Prerequisite Structures and Functions

The following are descriptions of major structures and functions that were implemented in order to create an efficient interface between implemented algorithms and the data supplied by the assignment.

3.1 Data Structures

This subsection addresses the underlying record structure of the implementation. Words on the efficiency of and justifications for these structures will be in subsection X.X.

3.1.1 Red-Black Trees

The main data structure used in this implementation is a red-black tree. A red-black tree is a self balancing form of a binary search tree. Red-black trees are responsible for storing the vertices, edges and open-queue (for breadth first search) in this implementation. The red-black tree is is defined by the following code, a reference to the root node of the tree. The record on line one is for the open queue and the record on line two is used for the vertices and edges.

```
1 (defrecord Red-Black-Tree [root])
2 (defrecord Red-Black-Tree-Map [root])
```

The root of the tree is a reference to a node. Tree nodes are defined by the code below. The two lines correspond to the open-queue, vertices and edges as with the red-black tree definitions. Both types of node keep track of the following

- "color": reference to the color of the node, either red or black, used to keep the tree balanced by related functions.
- "left": reference to the node that is to the left of the node.
- "right": reference to the node that is to the right of the node.
- "parent": reference to the parent of the node.
- "child": reference to the value of which child the node is of its parent (left, right, root).

The open-queue node stores the label and value of each node and uses the value to sort the nodes during insertion. The vertex or edge tree node stores "hashl", a hashed label of the vertex or edge, and "grecord", the record structure of the vertex or node (see 3.1.2 Graph). The vertex and edge trees are sorted based on the hashed labels.

```
(defrecord Red-Black-Node [label value color left right parent child])
(defrecord Red-Black-Map-Node [hashl grecord color left right parent child])
```

3.1.2 Graph

Graphs can be represented in many ways and in this implementation the author has chosen to use red-black trees with hash lookup to store a set of vertices and a set of edges. Each set has its own tree and these two trees make up the entire "Graph." record. The following code defines the graph's record structure.

```
(defrecord Graph [vertices edges])
```

Within the "grecord" references of the nodes within the red-black trees in either "vertices" or "edges" are "Vertex." or "Edge." record structures respectively. These records contain all data relevant to either the vertex or edge.

The folowing code defines the vertex and edge record structure. The supplied pieces of data for the vertex are "label", "neigbors", "latitude" and "longitude" though the neighbors are implied from the edges. The "status", "distance" and "component" of the vertex are utility data used by functions performing algorithms on the data. The "status" allows the breadth first search to check whether a vertex has yet been visited. The "distance" is where the distance from finish of the vertex is stored. Finanly "component" stores the indicie of the connected component to which the vertex belongs.

```
(defrecord Vertex [label neighbors latitude longitude status distance component])
(defrecord Edge [from to weight label])
```

The edge record structure contains only data supplied by the assignment.

3.2 Adding Data

The assignment supplied a list of statements that were meant to use the following functions:

```
(defn graph-add-vertex! [graph label latitude longitude]
  (let [hashed-label (hash-label label)]
    (when (not (red-black-hashmap-contains?
    hashed-label (:root @(:vertices graph))))
      (map-node-insert-helper-2!
        (:root @(:vertices graph))
        nil         hashed-label
        (Vertex. label (ref '()) latitude longitude
     (ref unseen) (ref ##Inf) (ref nil))
        false))))
```

```
(defn graph-add-edge! [graph from to label weight]
  (let [hashed-edge-key (hash-label (edge-key from to))]
    (when (not (red-black-hashmap-contains?
    hashed-edge-key (:root @(:edges graph)))))
      (map-node-insert-helper-2!
        (:root @(:edges graph))
        nil
        hashed-edge-key
        (Edge. from to weight label)
        true)))
```

These two functions

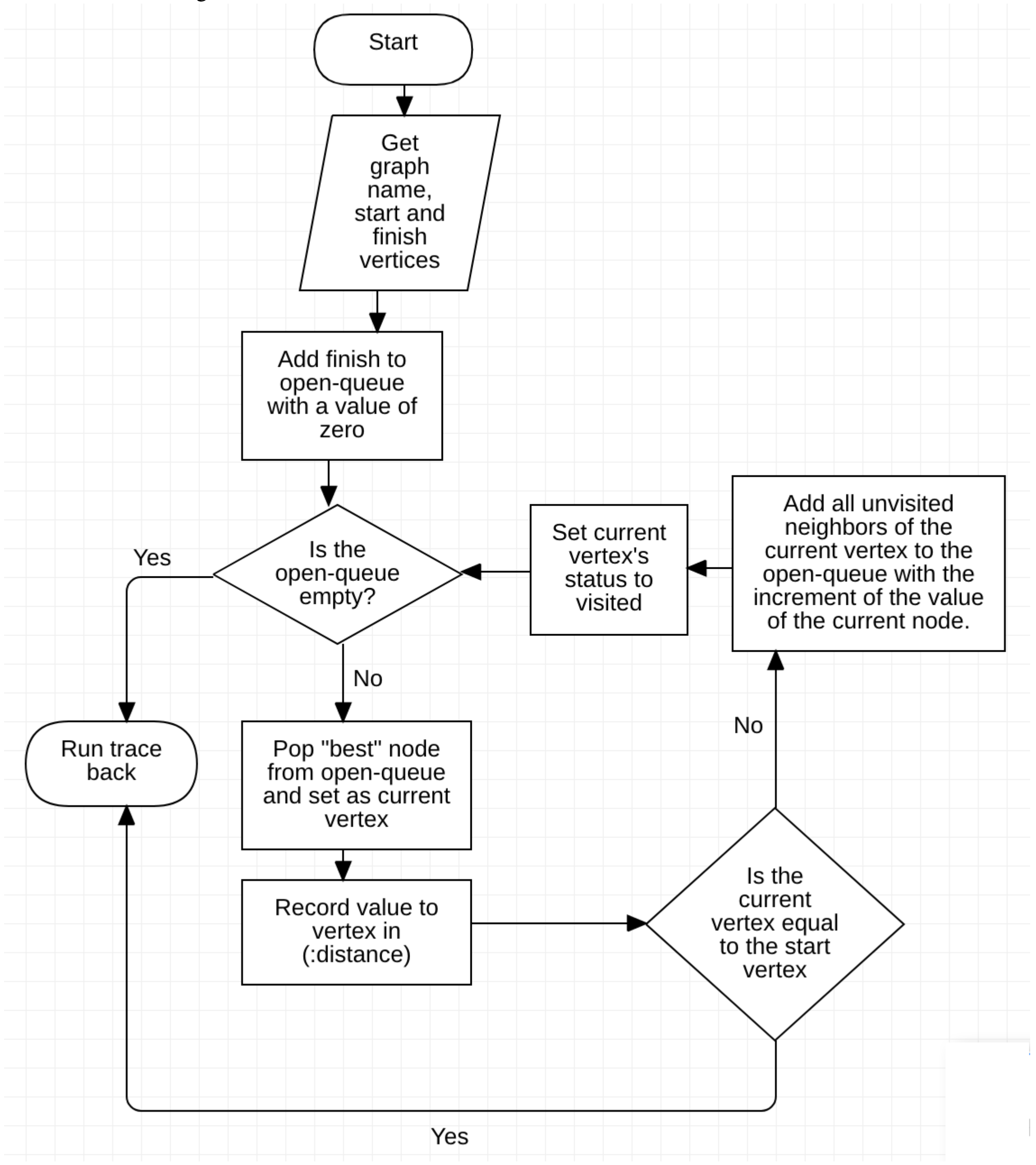
4 Dijkstra's Algorithm Without Edge Weights

This section describes the implementation of Dijkstra's algorithm, Dijkstra's algorithm finds the shortest path between two vertices in a graph. In this section only the implementation that does not take into account edge weights is discussed.

4.1 Breadth First Search Algorithm

```
(defn breadth-first-search-dijkstra [graph start finish]
  (node-insert! rb-queue finish 0)
  (loop []
    (when (not (red-black-tree-empty? rb-queue))
    (let [current (pick-least-node (:root rb-queue))]
      (remove-least-node! (:root rb-queue))
      (dosync
        (ref-set (:distance @(get-vertex graph (:label current)))
                 @(:value current)))
      (when (not (= (:label current) start))
        (loop [neighbors
               @(:neighbors @(get-vertex graph (:label current)))]
          (let [current-neighbor (first neighbors)]
            (when (get-vertex-unseen? graph current-neighbor)
              (node-insert! rb-queue current-neighbor (inc @(:value @current)))))
          (recur (rest neighbors)))))
    (recur))))
```

Figure 1: Flowchart for “breadth-first-search!” Function


Start
Get graph name, start and finish vertices
Add finish to open-queue with a value of zero
Is the open-queue empty?
Yes
No
Run trace back
Pop "best" node from open-queue and set as current vertex
Record value to vertex in (:distance)
Is the current vertex equal to the start vertex
No
Yes
Add all unvisited neighbors of the current vertex to the open-queue with the increment of the value of the current node.
Set current vertex's status to visited

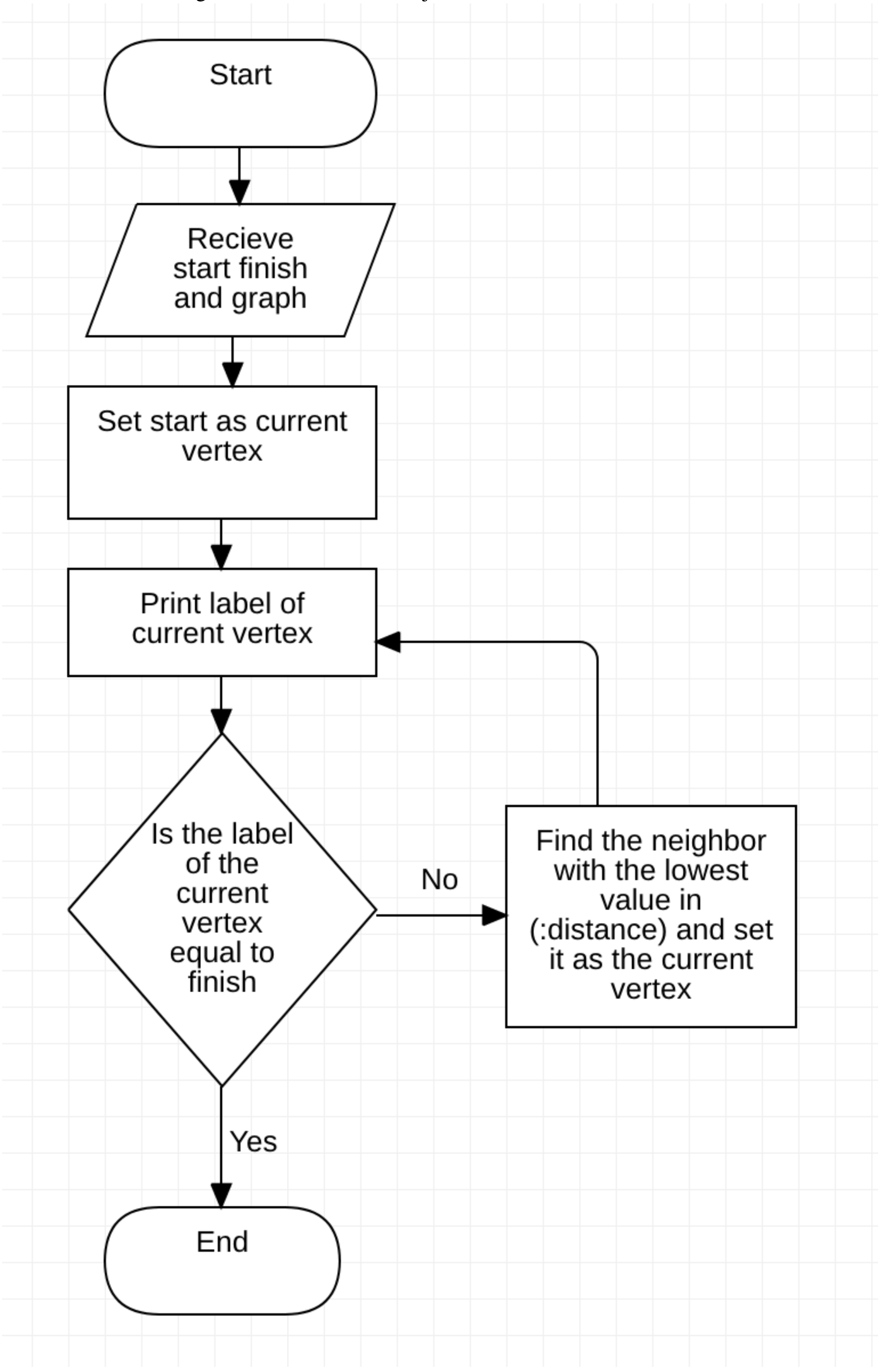
The algorithm above accepts three arguments one pointing to a graph within which are two nodes (start, finish) that are the remaining arguments. The algorithm finds the shortest path between these two nodes by running a breadth first search from the finish node searching for the start node. Breadth first search keeps track of an, “open queue” which is a set of vertices that are going to be processed, a set of visited or already proccessed vertices and a current vertex. The breadth first search is given the finish node which it adds to the open queue. If the open queue is still populated the algorithm removes next vertex from open queue and sets it as current vertex. The current vertex’s distance value (in its record structure) is set to its distance from the finish node. In this case the distance is the amount of hops (edges) the node is away from the finish. Finally all adjacent verticies of the current vertex that have an “unseen status” (in ther record structure) are added to the open queue and the current vertex’s status is set visited. The algorithm then choses its next vertex from the open queue and repeats the afformentioned steps untill it reaches the start node. It is at this point that the algorithm would normally evaluate whether or not a path exists between the two nodes however in this implementation each vertex has already been marked with an index number corresponding to its connected component. The algorithm first checks if the two nodes have equal values in the “component” sections of their records. If the indicies do not match there exists no path between the two nodes and the breadth first search will not

run. If this feature was not implemented the algorithm would have the itterate through the entire connected component before it could assume that there exists no path between the two given nodes.

4.2 Trace Back Function for Dijkstra's Algorithm

```
(defn dijkstra-trace-back [graph start finish]
  (loop [current start]
    (println current)
    (when (not (= current finish))
      (recur (dijkstra-trace-back-pick-best current)))))
```

Figure 2: Flowchart for “dijksta-trace-back” Function


Start
Recieve start finish and graph
Set start as current vertex
Print label of current vertex
Is the label of the current vertex equal to finish
No
Find the neighbor with the lowest value in (:distance) and set it as the current vertex
Yes
End

The above algorithm is used after the marking phase of Dijkstra’s algorithm without edge weights. The tracing back begins by setting the start vertex to be the current vertex. Next the algorithm prints the label of the current vertex. If the label of the current vertex is not equal to the finish argument the algorithm chooses the neighbor of the current vertex with the lowest distance value in its record and sets it to the new current vertex. After this the algorithm recurs back to the step where it prints the label and repeats stated steps until it reaches the finish. At this point Dijkstra’s algorithm has finished and the shortest path between the start and finish vertices has been printed.

4.2.1 Picking the Neighbor With The Lowest Distance To Finish

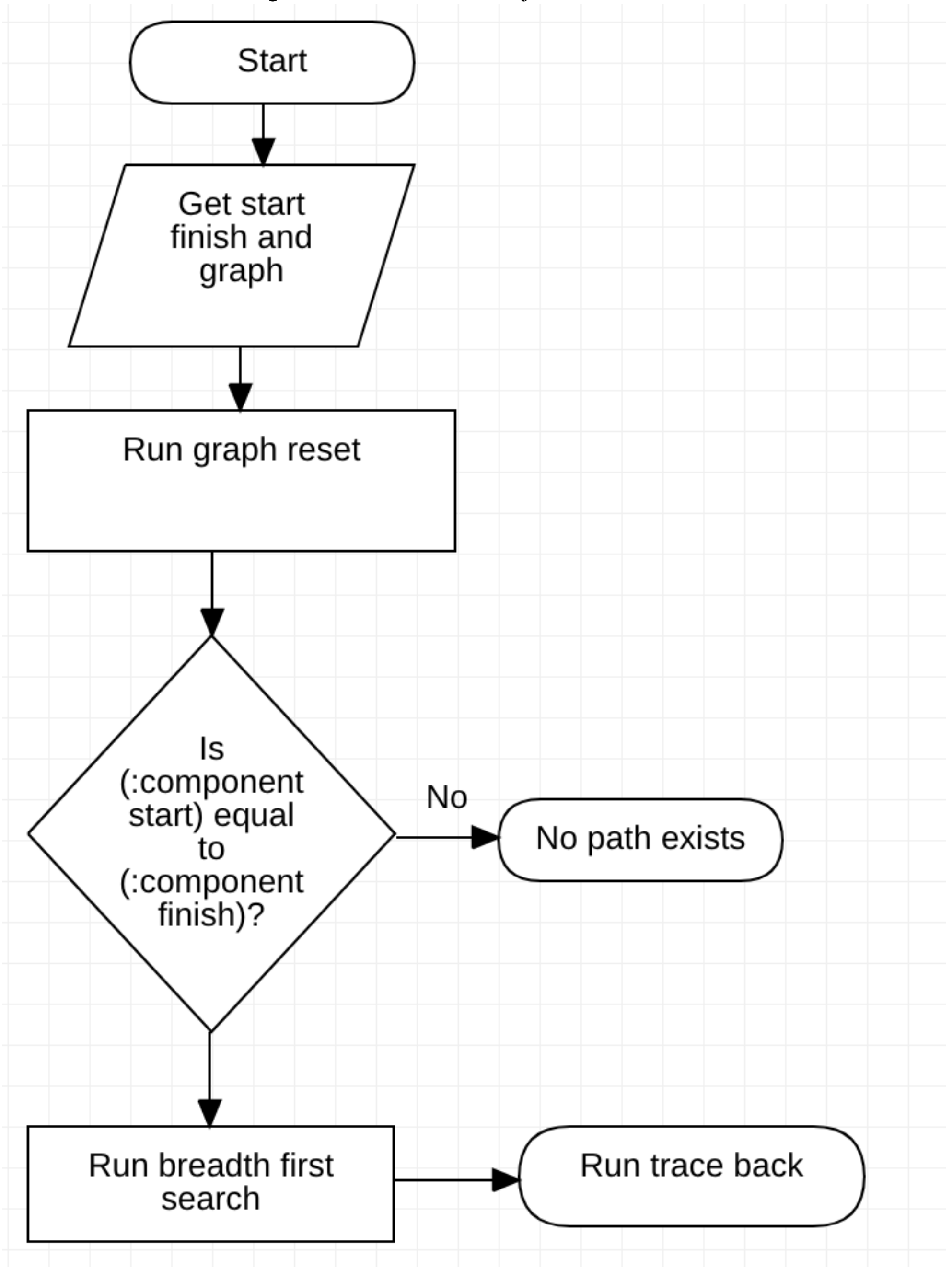
```
(defn dijkstra-trace-back-pick-best [graph vertex]
  (loop [neighbors @(:neighbors @(get-vertex graph vertex))
         best-distance ##Inf
         best-label nil]
    (if (= (count neighbors) 1)
      (if (< @(:distance @(get-vertex graph (first neighbors))) best-distance)
        (first neighbors)
        best-label)
      (if (< @(:distance @(get-vertex graph (first neighbors))) best-distance)
        (recur (rest neighbors)
               @(:distance @(get-vertex graph (first neighbors)))
               (first neighbors))
        (recur (rest neighbors)
                best-distance
                best-label)))))
```

Chosing the neighbor with the lowest distance to finish (without edge weights) is as simple as iterating through all the neighbors and keeping track of the lowest distance value over iterations. The above function acomplishes exactly that and returns the label of the neighbor with the lowest distance to finish. The time complexity of this function is tightly bound and linear O(n)= Ω(n)=Θ(n) where "n" is the number of neighbors of the vertex. This is because the distance each neighbor must be checked once to find the neighbor with the lowest value. The memory complexity of this function is not tightly bound O(n) Ω(1) as all of the neighbors must be stored in the loop at the beggining and one less is kept track off on each iteration.

4.3 Dijkstra Function

```
(defn dijkstra! [graph start finish]
  (graph-reset! graph)
  (if (= @(:component @(get-vertex graph start))
         @(:component @(get-vertex graph finish)))
    (do
      (breadth-first-search-dijkstra graph start finish)
      (dijkstra-trace-back graph start finish))
    (println "No␣path␣exists!")))
```

Figure 3: Flowchart for “dijkstra!” Function


Start
Get start finish and graph
Run graph reset
Is (:component start) equal to (:component finish)?
No
No path exists
Run breadth first search
Run trace back

The above function serves only to call all the helper functions associated with finding the shortest path between two vertices following Dijkstra’s algorithm. First the function runs the graph reset. This is a simple recursive function that iterates through all the vertices of the graph and sets their status to unseen and distance to finish to infinity. The time complexity of this operation is tightly bound and linear O(n)= Ω(n)=Θ(n) as it must iterate through each and every node of the red-black tree of vertices. The memory complexity of this varys throughout each iteration of the algorithm and is roughly equal to the width of the tree at the depth of the corresponding iteration this means it is tightly bound $O(2^i)= \Omega(2^i)=\Theta(2^i)$, where “i” is the index of the iteration starting at 0. This is however a rough estimate and may vary for leaf nodes as red-black trees stay within 1 node difference of maximum and minimum depth.

Next the function checks if the start and finish vertices are part of the same connected component. If they are, the function continues otherwise the function terminates without entering the marking phase as there exists no path between two nodes that are not part of the same connected component. The time complexity of this operation is O(log n) or Ω(1) and is not tightly bound because all that is happening is the lookup of two nodes within the graph’s red-black tree structure which contains the set of vertices. The

memory complexity is tightly bound and constant O(1)= $\Omega(1)=\Theta(1)$ as only one node is being accessed at a time while searching for the specified node.

If the condition is satisfied the function will call the marking stage which is a modified breadth first search. Once the marking stage completes the function runs the trace back algorithm and terminates.