Algorithms For Graph Structures

Martin Janda

June 8, 2020

Prague College

Algorithms and Data Structures (FT)

Dominik Pantůček

Shortest Route(s)

ICA2

3676 Words

Semester 2001

Contents

1 Scenario **1**

2 Aim **1**

3 Definitions **1**

4 Prerequisite Structures and Functions **1**
4.1 Data Structures 1
4.1.1 Red-Black Trees 1
4.1.2 Hash Maps 2
4.1.3 Adding Data 2
4.2 Justification 3

5 Indexing Connected Components **3**

6 Dijkstra's Algorithm Without Edge Weights **5**
6.1 Breadth First Search Function 5
6.2 Trace Back Function for Dijkstra's Algorithm 6
6.2.1 Picking the Neighbor With The Lowest Distance To Finish 8
6.3 Dijkstra Function 8
6.4 Efficiency and Justification 10

7 Finding the Shortest Path With Edge Weights **10**
7.1 Breadth First Search for A* Function 10
7.2 Weighted Trace Back Function for A* 11
7.2.1 Picking the Valid Neighbor With the Lowest Distance to Finish . . . 13
7.3 A* Function 13
7.4 Efficiency and Justification 14

8 Conclusion **14**

9 References **14**

List of Figures

1 Flowchart for "breadth-first-search!" Function 6
2 Flowchart for "dijkstra-trace-back" Function 7
3 Flowchart for "dijkstra!" Function 9
4 Flowchart for "breadth-first-search-a*!" Function 11
5 Flowchart for "wighted-trace-back" Function 12

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 this assignment is to describe the most efficient implementation of the algorithms specified by the assignment brief. This technical report also addresses the graph and helper structures that have been implemented as they are a large part of the code being submitted and contribute directly to the efficiency of the algorithms. All code in this document was written by the author in Clojure a list processing programming language.

3 Definitions

- Graph: A set of vertices and edges.
- Sparse (graph): A graph that has less then v^2 edges, where "v" is the number of vertices.
- Path: A sequence of vertices connected by edges. (Pantůček, 2020)
- Cycle: A path that begins and ends on the same vertex.
- Tree: A form of graph that contains no cycles.

4 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.

4.1 Data Structures

This subsection addresses the underlying record structures of the implementation. Words on the efficiency of and justifications for these structures are in subsection 4.2.

4.1.1 Red-Black Trees

The data structure used for the open-queue in this implementation is a red-black tree. A red-black tree is a self balancing form of a binary search tree. The red-black tree is is defined by the following code, a reference to the root node of the tree.

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

The root of the tree is a reference to a node. Tree nodes are defined by the code below and keep track of the following:

- "label": the label of the node.
- "value": the value by which the node is sorted.
- "value2": the second value used to store distance from finish when "value" is used for great circle distance in A*.
- "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).

```
1 (defrecord Red-Black-Node [label value value2 color left right parent child])
```

4.1.2 Hash Maps

Clojure's built in hash map data structure is used to store all the nodes and vertices of the graph. The graph itself is a record that contains references to two hash maps, one for the vertices and one for the edges.

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

4.1.3 Adding Data

When adding nodes to the open-queue the function "node-insert!" is called.

```
1  (defn node-insert! [tree label value value2]
2    (if (red-black-tree-empty? tree)
3      (dosync
4        (ref-set (:root tree)
5          (make-node! label value value2 Black nil Root)))
6      (cond
7        (< value @(:value @(:root tree)))
8          (node-insert-helper! (:left @(:root tree)) (:root tree)
9          label value value2 Left)
10       (> value @(:value @(:root tree)))
11         (node-insert-helper! (:right @(:root tree)) (:root tree)
12            label value value2 Right)
13       (= value @(:value @(:root tree)))
14         (node-insert-helper! (:right @(:root tree)) (:root tree)
15         label value value2 Right))))
```

```
(defn node-insert-helper! [node parent label value value2 child]
  (if (tree-node-empty? node)
    (do
      (dosync
        (ref-set node
          (make-node! label value value2 Red @parent child)))
      (red-black-rules-checker! node))
    (cond
      (< value @(:value @(:root tree)))
        (node-insert-helper! (:left @(:root tree)) (:root tree)
        label value value2 Left)
      (> value @(:value @(:root tree)))
        (node-insert-helper! (:right @(:root tree)) (:root tree)
           label value value2 Right)
      (= value @(:value @(:root tree)))
        (node-insert-helper! (:right @(:root tree)) (:root tree)
        label value value2 Right))))
```

These functions follow the standard procedure for inserting a node into a binary search tree. If the node being inserted is not the root of the red-black tree the "node-insert-helper!" function is called and it runs the "red-black-rules-checker!" function after the insertion. The "red-black-rules-checker!" function checks if the tree is still a red-black tree and performs all necessary rotations and color changes necessary to balance and restore its state.

4.2 Justification

Using a red-black tree has a number of benefits. Using a list for the open-queue in breadth first search his not optimal. In Clojure lists are single-linked which means that the head of the list contains a reference to only the first node and nodes only contain a reference to the next node and not the previous. This means that appending to the list would have a tightly bound linear time complexity O(n)= Ω(n)=Θ(n). Furthermore accessing the node with the lowest value would have a time complexity of O(n) and for the best case scenario Ω(1), as the list is not sorted.

The red-black tree solves this problem as it is a form of a binary search tree. The tree is sorted and thus accessing the node with the lowest value is as simple as going to the left-most node of the tree. The time complexity of this is tightly bound O(log n)= Ω(log n)=Θ(log n) and the memory complexity is constant. For insertions the time complexity is not tightly bound O(log n) and Ω(1) if the tree is empty. This logarithmic time complexity is due to the maximum height of any red-black tree which is O(log n) thanks to the self balancing nature of the tree.

It was considered that the entire graph structure could be represented using red-black trees. However, the efficiency of this could not be justified as Clojure has a hash map structure that has a near constant time complexity for accessing and inserting nodes.

5 Indexing Connected Components

In each vertex record structure is as "component" value. This value is used to tell whether any two vertices are part of the same component. This prevents algorithms like Dijkstra's algorithm from having to iterate through an entire connected component if there exists no path between the start and finish vertex. The result of this is used to

prove there exists no path between two vertices (in this code "Palmero" and "Prague"). Below are the functions involved.

```
(defn breadth-first-search-connected-components! [graph start index]
  (node-insert! rb-queue start index nil)
  (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 (:component (get-vertex graph (:label current)))
                 @(:value current)))
      (loop [neighbors
             @(:neighbors (get-vertex graph (:label current)))]
        (let [current-neighbor (first neighbors)]
          (when (vertex-unseen? graph current-neighbor)
            (node-insert! rb-queue current-neighbor index nil)))
        (recur (rest neighbors))))
      (dosync
        (ref-set (:status (get-vertex graph (:label current))) visited))
    (recur))))
```

The "breadth-first-search-connected-components!" function accepts 3 arguments graph, start and index. The function iterates through the entire connected component of the start vertex. First the algorithm adds the start vertex to the open-queue. While the queue is not empty it sets the left-most node of the red-black tree open-queue as the current vertex. Then the function sets the component value of the vertex equal to the index value passed to the function. Finally it adds all the unseen neighbors of the current vertex to the open-queue, sets the status of the current vertex to visited and recurs to the stage of checking whether the open-queue is empty. This function is called by the main indexing function below.

```
(defn index-components! [graph]
  (graph-reset! graph)
  (loop [index 0]
    (breadth-first-search-connected-components! graph
                                                (remaining-unseen graph)
                                                index)
    (recur (inc index))))
```

The "index-components!" function is a simple tail recursive loop that keeps track of the index value that is going to be passed to the breadth first search. First the status and distance values of the graph are reset. Then the loop calls the "breadth-first-search-connected-components!" using the first "unseen" vertex it finds as the start vertex. The loop recurs with the increment of the index and runs again. The loop fails when it can no longer find an "unseen" vertex in the graph.

The time complexity of this function is tightly bound and linear O(n)= Ω(n)=Θ(n) as the breadth first search will iterate through the entire graph before the loop ends. The memory complexity of this function is tightly bound and constant O(1)= Ω(1)=Θ(1) as it only keeps track of a set amount of nodes at a time. This is theoretically the only way to label the connected components of a graph as regardless of the traversing function

the entire set of vertices must be iterated through. Thanks to this being run after all the data has been added to the graph, the time complexity of proving there exists no path between two vertices is near constant due to Clojure's hash maps.

6 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.

6.1 Breadth First Search Function

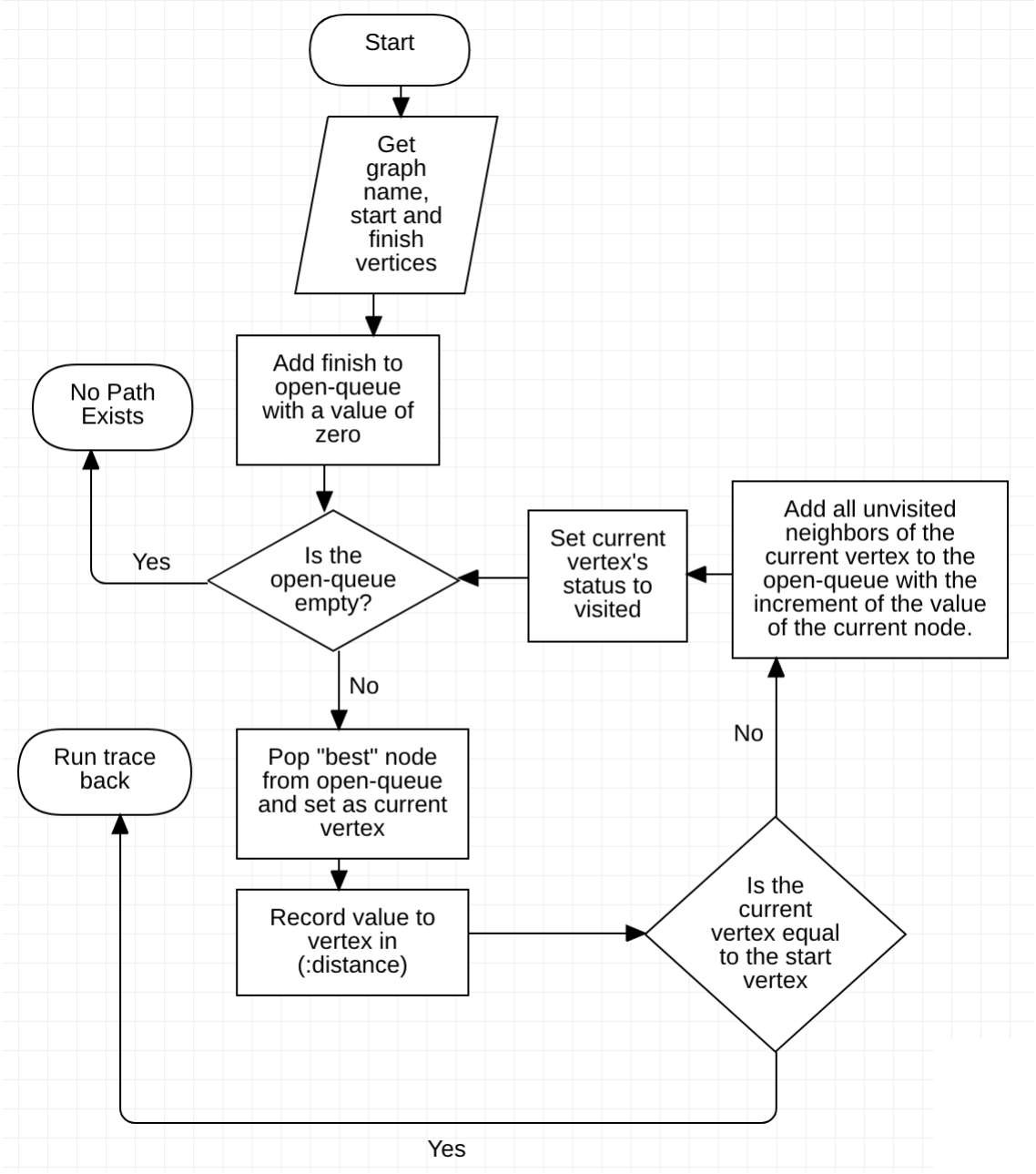
```
1  (defn breadth-first-search-dijkstra! [graph start finish]
2    (node-insert! rb-queue finish 0 nil)
3    (loop []
4      (when (not (red-black-tree-empty? rb-queue))
5      (let [current (pick-least-node (:root rb-queue))]
6        (remove-least-node! (:root rb-queue))
7        (dosync
8          (ref-set (:distance (get-vertex graph (:label current)))
9                   @(:value current)))
10       (when (not (= (:label current) start))
11         (loop [neighbors
12                @(:neighbors (get-vertex graph (:label current)))]
13           (let [current-neighbor (first neighbors)]
14             (when (vertex-unseen? graph current-neighbor)
15               (node-insert! rb-queue current-neighbor (inc @(:value current)) nil)
16           (recur (rest neighbors)))))
17       (dosync
18         (ref-set (:status (get-vertex graph (:label current))) visited))
19     (recur))))
```

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 processed vertices and a current vertex.

The breadth first search is given the finish node which it adds to the open queue along with a value of zero. While 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 the graph's record structure) is set to the value attributed to it in the open queue. In this case the distance is the amount of hops (edges) the node is away from the finish. Finally all adjacent vertices of the current vertex that have an "unseen" status (in the graph's record structure) are added to the open queue with the increment of the value of the current vertex. The current vertex's status is then set visited.

The algorithm then choses its next vertex from the open queue and repeats the aforementioned steps until it reaches the start node. The breadth first search will iterate through the entire connected component if it does not reach the start node. It is at this point that the "dijkstra!" function 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.

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



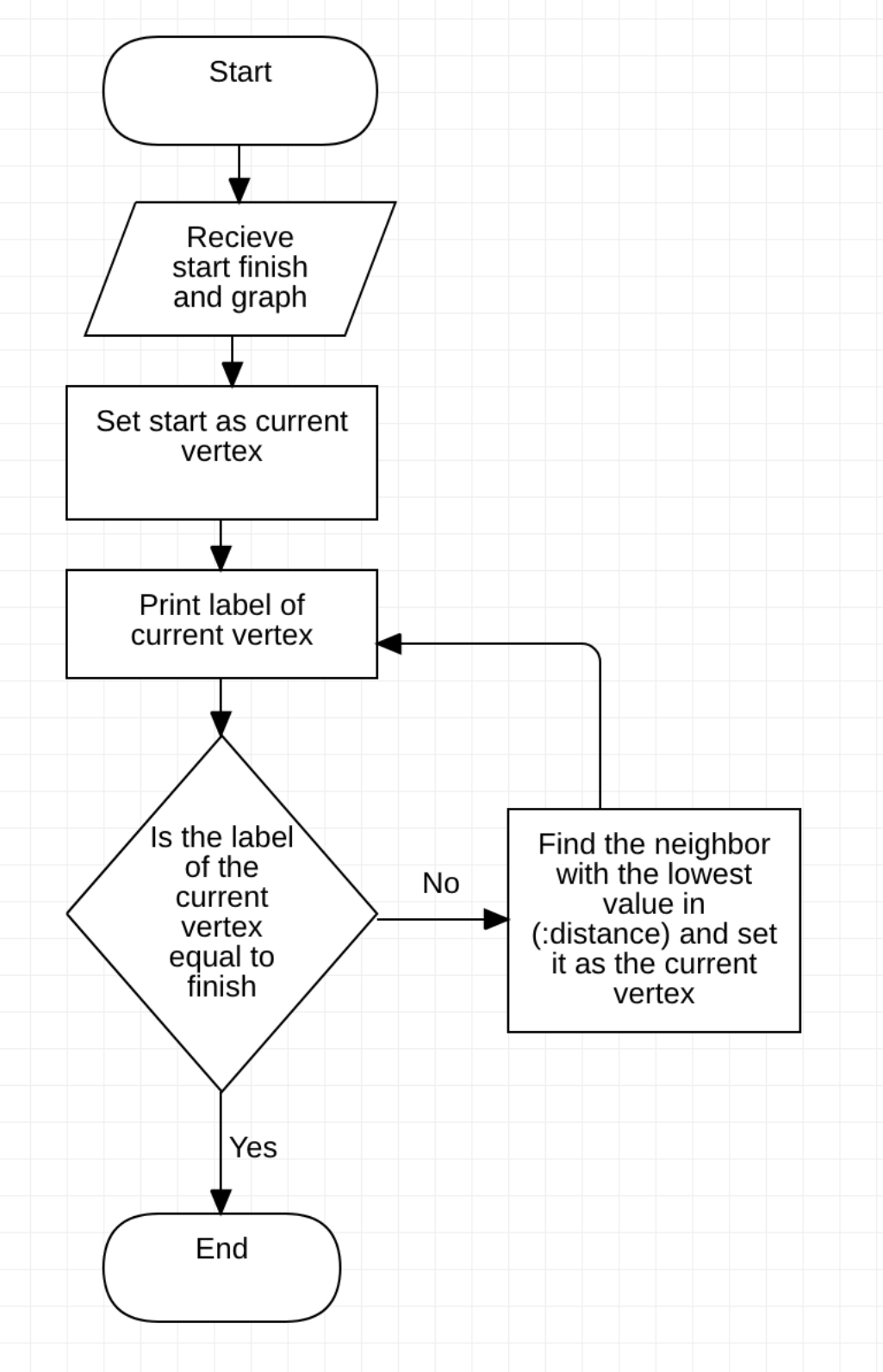
The “dijkstra!” function first checks if the two nodes have equal values in the “component” section of their records. If the indices do not match there exists no path between the two nodes and the breadth first search will not be run. If this feature was not implemented the breadth first search algorithm would have the iterate through the entire connected component before it could assume that there exists no path between the two given nodes.

6.2 Trace Back Function for Dijkstra’s Algorithm

```
1 (defn dijkstra-trace-back [graph start finish]
2   (loop [current start]
3     (println current)
4     (when (not (= current finish))
```

```
5          (recur (dijkstra-trace-back-pick-best current)))))
```

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



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.

6.2.1 Picking the Neighbor With The Lowest Distance To Finish

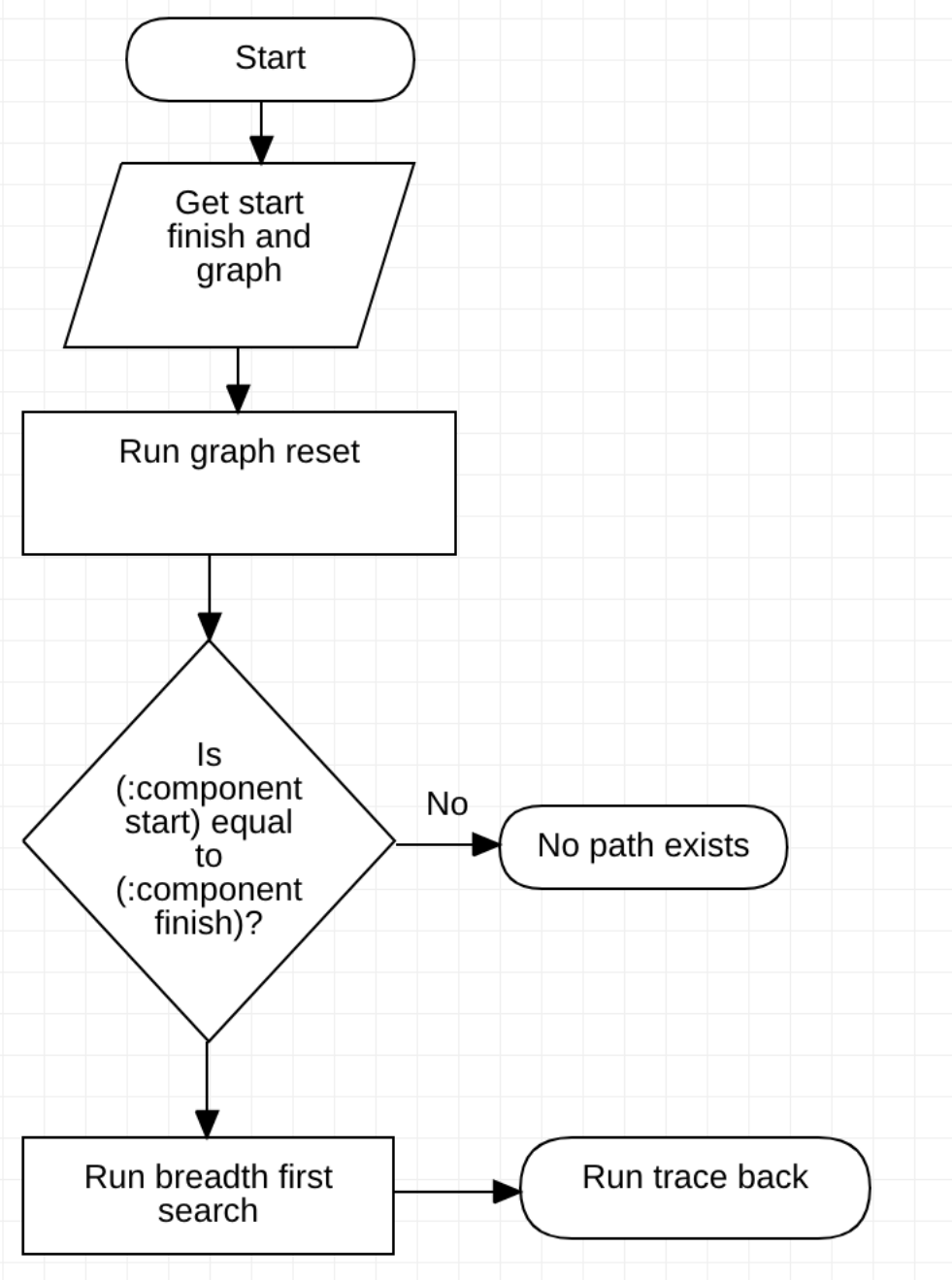
```
1  (defn dijkstra-trace-back-pick-best [graph vertex]
2    (loop [neighbors @(:neighbors @(get-vertex graph vertex))
3           best-distance ##Inf
4           best-label nil]
5      (if (= (count neighbors) 1)
6        (if (< @(:distance @(get-vertex graph (first neighbors))) best-distance)
7          (first neighbors)
8          best-label)
9        (if (< @(:distance @(get-vertex graph (first neighbors))) best-distance)
10         (recur (rest neighbors)
11                @(:distance @(get-vertex graph (first neighbors)))
12                (first neighbors))
13         (recur (rest neighbors)
14                 best-distance
15                 best-label)))))
```

Choosing 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 accomplishes 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 beginning and one less is kept track off on each iteration.

6.3 Dijkstra Function

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

Figure 3: Flowchart for “dijkstra!” Function



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.

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 nearly constant because all that is happening is the lookup of two nodes within the graph’s hash map structure which contains the set of vertices.

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.

6.4 Efficiency and Justification

Aside from possible micro-optimizations this is the most efficient way to find the shortest path between two point on a graph. This algorithm has been chosen simply because there is no other algorithm to find the shortest path between two points in an unweighted graph.

Several optimizations have already been added to this implementation. The worst case scenario for naive Dijkstra's algorithm occurs when there exists no path between the vertices supplied. With this implementation if the API gets more than "x" (where "x" is the amount of connected components) requests for nonexistent paths it will be more efficient. This is because the connected components are counted and marked after the entire graph has been recorded. The open-queue has also been optimized using a red black tree. The time complexity of this function is not tightly bound $O(n^2)$ and $\Omega(1)$ in a sparse graph.

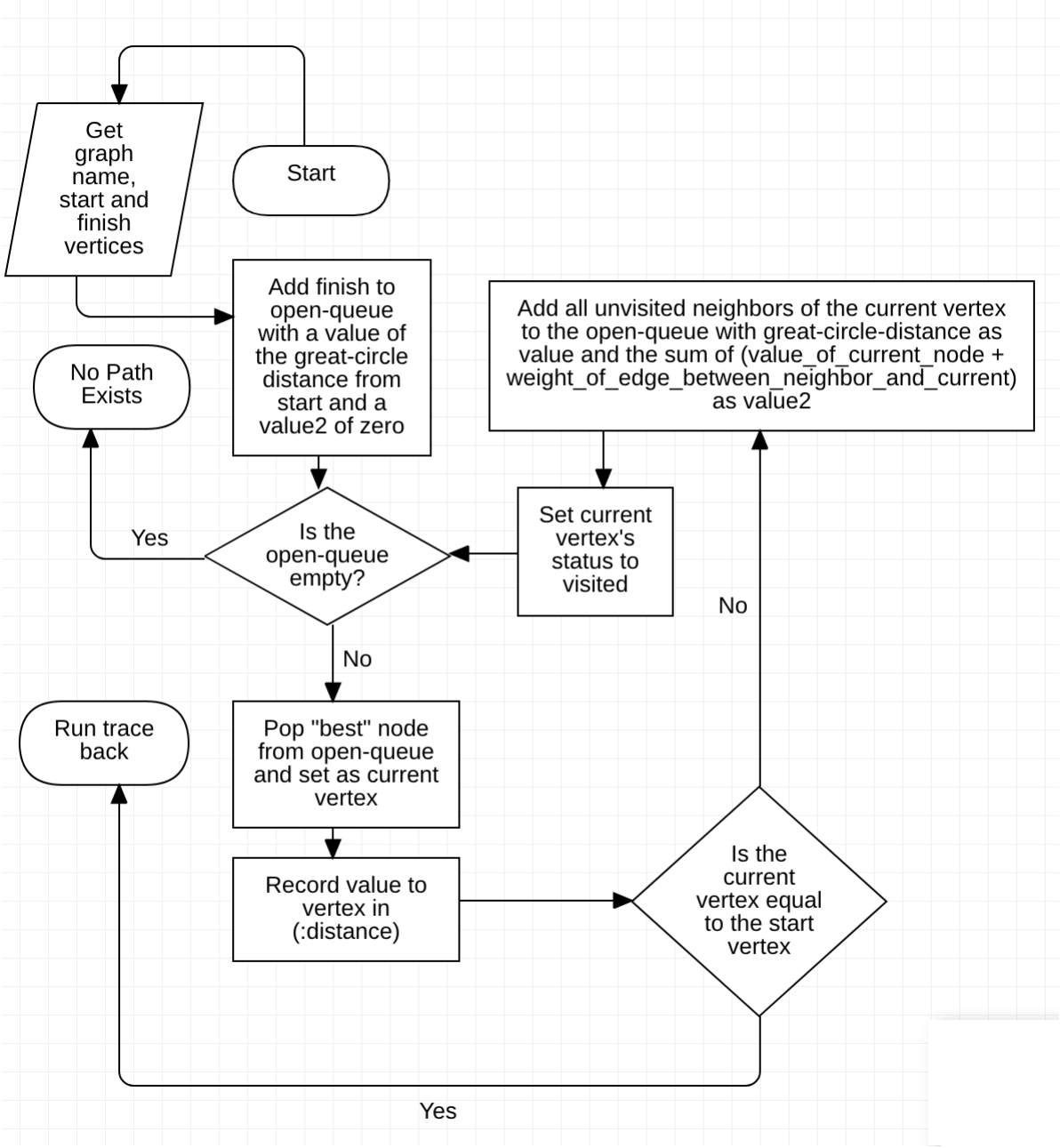
7 Finding the Shortest Path With Edge Weights

This section discusses algorithms for finding the shortest path between two vertices in a weighted graph.

7.1 Breadth First Search for A* Function

```
1  (defn breadth-first-search-a*! [graph start finish]
2    (node-insert! rb-queue finish (great-circle-distance graph finish start) 0)
3    (loop []
4      (when (not (red-black-tree-empty? rb-queue))
5      (let [current (pick-least-node (:root rb-queue) start)]
6        (remove-least-node! (:root rb-queue))
7        (dosync
8          (ref-set (:distance (get-vertex graph (:label current)))
9                   @(:value2 current)))
10       (when (not (= (:label current) start))
11         (loop [neighbors
12                @(:neighbors (get-vertex graph (:label current)))]
13           (let [current-neighbor (first neighbors)
14                 edge-distance (:distance (get-edge graph
15                                                    current-neighbor
16                                                    (:label current)))
17                 great-circle (great-circle-distance graph
18                                                     current-neighbor
19                                                     start)]
20             (when (vertex-unseen? graph current-neighbor)
21               (node-insert! rb-queue current-neighbor
22                             great-circle
23                             (+ @(:value current) edge-distance))))
24           (recur (rest neighbors)))))
25     (recur))))
```

Figure 4: Flowchart for “breadth-first-search-a*!” Function

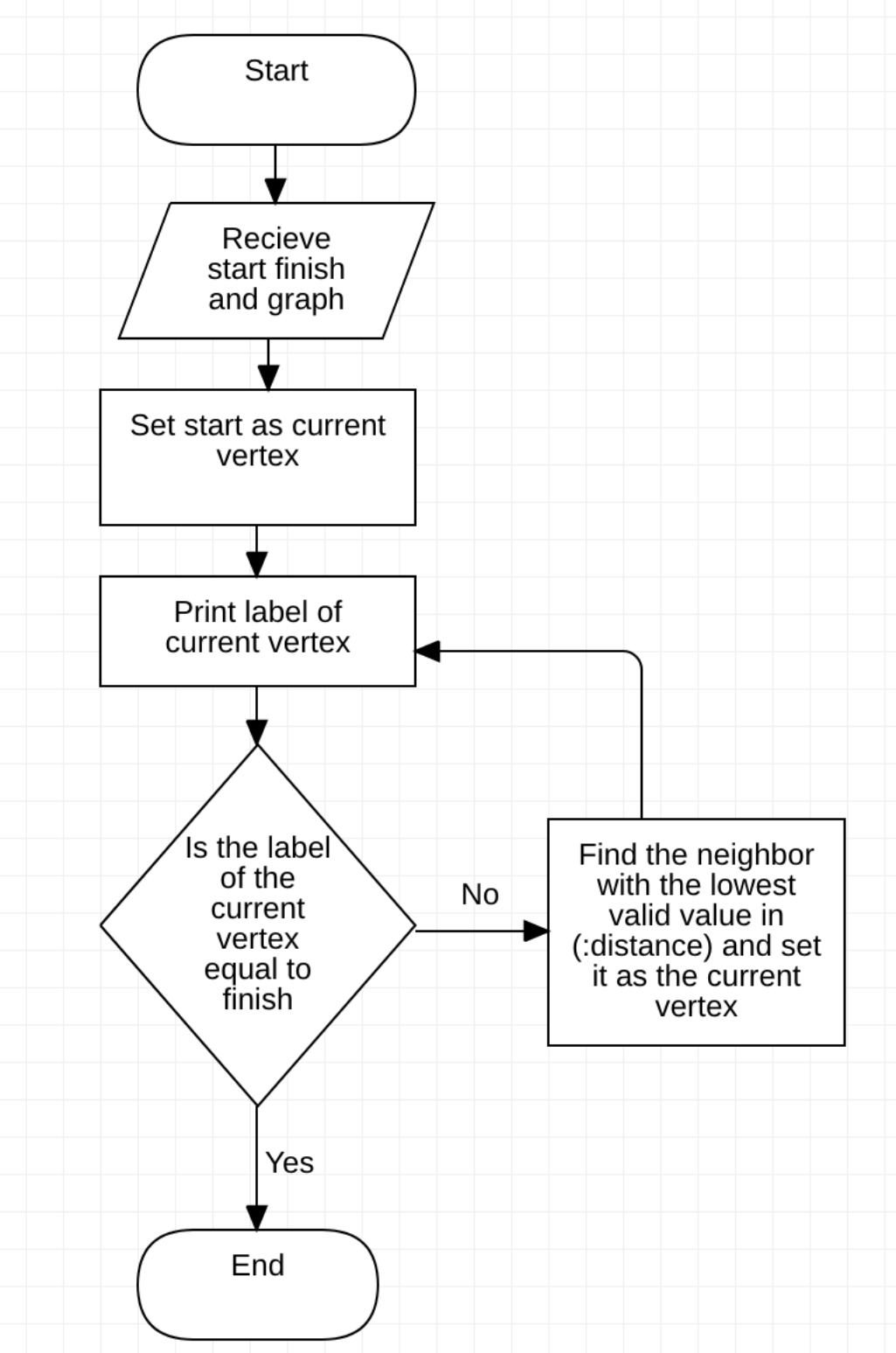


The “breadth-first-search-a*!” function follows the same steps as any other breadth first search except it uses the great-circle distance of the vertex to the start to pick the next vertex to the open-queue. This is accomplished by storing the great-circle distance in the “value” section of the red-black tree nodes record, the edge weight distance is stored in “value2”.

7.2 Weighted Trace Back Function for A*

```
1 (defn weighted-trace-back [graph start finish]
2   (loop [current start]
3     (println current)
4     (when (not (= current finish))
5       (recur (weighted-trace-back-pick-best current)))))
```

Figure 5: Flowchart for “wighted-trace-back” Function



After the marking stage of A* algorithm has finished the “weighted-trace-back” function is run. This function is capable of tracing the shortest path marked by both A* are Dijkstra with weights. Like its unweighted counterpart it accepts three arguments graph, start and finish. The function follows the exact same steps as its unweighted counterpart as well, the only difference is the function it uses to find the neighbor with

the lowest value.

7.2.1 Picking the Valid Neighbor With the Lowest Distance to Finish

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

In order to choose the neighbor with the lowest distance to finish this function must also validate that the distance of this vertex is correct. This is due to the nature of the marking stage of A*. The "weighted-trace-back-pick-best" function follows the same steps as the "dijkstra-trace-back-pick-best" function save for one detail. When deciding if a neighbor has a lower distance than the current distance the function checks if difference between the distance values of the neighbor and the vertex is equal to weight of the edge connecting them. If the previous condition is true and the distance is lower than the best distance the function either recurs with or returns the label of the neighbor being evaluated.

7.3 A* Function

```
(defn a*! [graph start finish]
  (graph-reset! graph)
  (def rb-queue (make-red-black-tree!))
  (if (= @(:component @(get-vertex graph start))
         @(:component @(get-vertex graph finish)))
    (do
      (breadth-first-search-a*! graph start finish)
      (weighted-trace-back graph start finish))
    (println "No path exists!")))
```

The "a*!" function follows the same exact steps as pictured in Figure 3. The only difference is the function calls the A* versions of breadth first search and trace back functions. A* is in principle a version of Dijkstra's algorithm that is capable of "aiming" its search towards the start node. This generally makes it more efficient.

7.4 Efficiency and Justification

A* was chosen because it is one of the most efficient algorithms for finding a path in a weighted graph. In fact there are very few scenarios in which a different algorithm would outperform A*. In a trivially small graph Dijkstra's algorithm would outperform A* because they will both do the same amount of iterations. If multiple search operations are to be performed all coming from the same starting vertex a slightly modified Dijkstra's algorithm would work much better.

A* uses heuristic values (latitude and longitude) in a type of "best" first search (Pantůček 2020), since the data supplied contained latitude and longitude values for each vertex A* made the most efficient use of the data supplied. The time complexity of A* has a direct relationship with the heuristic function, it is not tightly bound and generally is not written as a function of the number of vertices in a graph. Speculations can be made that it is exponential for the length of the path returned in the upper bound (Santoso, 2010).

8 Conclusion

In general the subject of graph-related algorithms is an extremely nuanced part of mathematics in computing. There are countless real world applications, even for the algorithms discussed in this document (networking, GPS Navigation, logistics). The authors goal when writing this code was to write the most efficient versions of these algorithms within his capabilities. However, this is in no way a claim that there aren't any improvements to be made. With higher level functions such as this there are so many variables to change and optimize (data structure, helper functions) and due to this there is always some angle that has not been considered. It is the driving force behind competition and improvement in software over the years. What this assignment did accomplish was the delivery of a clear competent explanation of the given algorithms and a justification for the improvements that have been made.

9 References

Pantůček, D., 2020, Algorithms and Data Structures (ALDS-400-2001), Spring 2020 21. A* Search Algorithm Finding Shortest Path in Weighted Graph, Presentation, Prague College, Prague.

Pantůček, D., 2020, Algorithms and Data Structures (ALDS-400-2001), Spring 2020 17. Dijkstra's Algorithm Finding Shortest Path, Presentation, Prague College, Prague.

Santoso, L. W., 2010, Performance Analysis of Dijkstra, A* and Ant Algorithm for Finding Optimal Path Case Study: Surabaya City Map, Article, Petra Christian University, Surabaya.