1 Project Overview

1.1 Brief Description

This application is a software for a delivery company that can help people schedule the delivery of items so that the one with the soonest deadline will be delivered first. People could also use it to find the information of a specific item using the item ID. This software could also help people find the shortest path from the warehouse to the destination place of a given item.

1.2 Assumptions

1. Item information, including its ID, delivery deadline, location in the warehouse, and destination place, is stored in a the Item Class.

2. All item information is stored as private fields, so there are corresponding public getter functions to get these information out of the Item class.

3. If we say two items are the same, their ID, delivery deadline, location, and destination all need to be the same.

4. The test input files are divided into two categories, one is about item information, which will be used in function 1 and 2. The other one is about graph information, which will be used in function 3 only.

5. The graph we used is a directed weighted graph, and all edges' weights are positive numbers.

6. The test file about item information follows format: item ID, deadline, location, destination. And the test file about graphs follows format: edge start point, edge end point, edge weight.

7. Information about people who deliver the item will not be part of this application.

8. The number of items in the test file I used to do the empirical analysis is in the range $[2^6, 2^{26}]$, but only five of them have been attached to the zip file submitted to wattle. You could run the generateData program to generate on your own or download it from GitLab.

9. The number of vertices in the graph file I used to do the empirical analysis is in the range $[2^3, 2^{16}]$, but only five of them have been attached to the zip file submitted to wattle. You could run the generateData program to generate on your own or download it from GitLab.

1.3 Functionalities for Marking

Function 1 - Get the item with the soonest deadline by scheduling the orders using min heap.

Role: This function will be used when people want to find the item with the soonest deadline so that people can work on the most urgent one.

Function 2 - Find the item information with given ID using Red Black trees.

Role: This function will be used when people want to quickly find where an item is placed in the warehouse (i.e. location) or where this item will be sent to (i.e. destination). This will be performed quite frequently when people manage the warehouse.

Function 3 - Identify the shortest path from the warehouse to the destination place of an item using Dijkstra's algorithm.

Role: This function will be used when people want to schedule a shortest way for a driver to deliver the item, which could save both money and time.

2 Function 1: Get the item with the soonest deadline

2.1 Data Structure: Min-Heap

The reason I used min-heap here is mainly because we would like to get the item with the soonest deadline as fast as we can. The deadline information is stored as a positive number in the test file. So the most suitable data structure is min-heap. Getting the item with the minimum deadline number only cost constant time (time

complexity $O(1)$). And the extract-min function has a corresponding time complexity $O(n)$. Meanwhile the time complexity for insertion operation in min-heap is $O(\log n)$, so it can also be built up in a short time and be maintained easily.

2.2 Theoretical Time Complexity Analysis

Min-heap is a complete binary tree, each node has a key and some additional information, in this case, is the Item object. All items are stored in an array. The key value used in this project is the deadline date of the item, and since it is a min-heap, each parent node has a smaller key than its child nodes. Thus, the smallest key means the soonest deadline, and if we want to get the item with the soonest deadline, we just need to call the peek method to return the first element in the array. But to be able to do so, we need first have all elements stored in the array in the right order, which based on the insert and swim methods. Also, in the real world, if we get the item with the soonest deadline and process that order, we need to remove it from the warehouse, thus we need to call the extractMin and sink method to remove that item in heap.

Overall, I did theoretical analysis on the following methods: more, swap, swim, insert, sink, extractMin, build, and peek.

2.2.1 more and swap

These two methods are helper functions used in the MinHeap class. The more method compare which item has a sooner deadline. Since it only has a if statement, the time complexity of the line 2 is $O(1)$, in both average and worst case. The swap method swaps two items inside the array, lines 5 to 7 are all constant statements, thus the time complexity of the swap methods is also $O(1)$, in both average and worst case.

```
1  private boolean more(int i, int j) {
2      return items[i].getDeadline() > items[j].getDeadline();
3  }
4  public void swap(int i, int j) {
5      Item temp = items[i];
6      items[i] = items[j];
7      items[j] = temp;
8  }
```

2.2.2 swim and insert

The running time of the insert operation is the combination of running a constant amount of work (line 2) and running the swim operation.

The parameter k in the swim method is the size of the array, i.e. the total number of elements in the heap. $k/2$ here means the parent node of the node k. In the worst case, every time the parent node is larger than the current node, thus we move into the while loop and do two constant operations (line 7 and 8). Since $2^{height} = n$, we have the height of the heap is $\log n$. So in the worst case, we will do swapping for $\log n$ times, which means the running time of the swim operation in the worst case is $O(\log n)$. And the running time under the average case, which means we will do the swapping operations for $\frac{1}{2}\log n$, is still $O(\log n)$.

Overall, the running time of swim and insert operation is $O(\log n)$.

```
1  public void insert(Item newItem) {
2      items[++size] = newItem;
3      swim(size);
4  }
5  private void swim(int k) {
6      while (k > 1 && more(k / 2, k)) {
7          swap(k, k / 2);
8          k = k / 2;
9      }
10 }
```

2.2.3 sink and extractMin

Similar to insert, the running time of extractMin is the combination of running a constant amount of work (line 2-5) and running the sink operation.

The parameter k in the swim method is 1, i.e. the element on top of the heap. $2 * k$ here means the left child node of the node k. In the worst case, each internal node has a left child node and the left child node is always smaller than the current internal node, which means we need to do the swap operation, since the height of the heap is $\log n$, and the running time from line 11 to 15 are all constant, the running time of sink in the worst case is $O(\log n)$. Similar to insert, in the average case, the running time is also $O(\log n)$.

Overall, the running time of sink and extractMin operations is $O(\log n)$.

```
1  public Item extractMin() {
2      if (size == 0) throw new IllegalStateException("Heap is empty.");
3      Item min = items[1];
4      swap(1, size--);
5      items[size + 1] = null;
6      sink(1);
7      return min;
8  }
9  private void sink(int k) {
10     while (2 * k <= size) {
11         int j = 2 * k;
12         if (j < size && more(j, j + 1)) j++;
13         if (!more(k, j)) break;
14         swap(k, j);
15         k = j;
16     }
17 }
```

2.2.4 build

The time complexity of the build operation seems like $O(n \log n)$, since it runs the sink operations for $\frac{n}{2}$ times, and the time complexity of sink is $O(\log n)$. But this is not tight enough, because the running time of sink at a node varies with the height of that node in the heap. Since the height of heap is $\log n$ and each level has at most $\lceil n/2^{h+1} \rceil$, where n is the total number of nodes in the heap and h is the height. Then we have

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h})$$

Since, $\sum_{h=0}^{\infty} h/2^h = 2$, we have

$$O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

Thus, the build operation time complexity is $O(n)$.

```
1  public void build() {
2      for (int i = size / 2; i >= 1; i--) {
3          sink(i);
4      }
5  }
```

2.2.5 peek

The peek method just return the first element in the array items, thus the time complexity is $\Theta(1)$.

```
1  public Item peek() {
2      return items[1];
3  }
```

2.3 Empirical Time Complexity Analysis

2.3.1 Methodology

Since Java Virtual Machine is a managed runtime environment, a lot of factors could affect the code performance, such as the garbage collector, the Just in Time compilation, and the heap size. Thus, to minimize

the impact of these factors, I followed the approach given by Costa et al. (2017) and followed a three-step methodology to do the empirical analysis:

- Warm up: Do ten warm-up iterations to achieve a steady performance.
- Execute 100 iterations to measure the time of the operation, and calculate the average execution time.
- Repeat the second step for two more times to avoid circumstantial external influence.

Note: This approach has also been used in the following empirical time complexity analysis for the other two functions.

2.3.2 Data

Data used in the analysis below is generated by a random data generator written by myself. The basic structure is: each parcel is an item object with a distinct item ID, delivery deadline, location in the warehouse, and destination place code. All item data is stored in a txt file, while each line represent an item whose attribute values are separated by a comma.

The data set size increase from 2^6 to 2^{26}. The reason for choosing such a range is because the extract-Min operation has a $O(\log n)$ time complexity.

2.3.3 Results

As shown in the figure 1, the red points are the data points generated after running the program for several times. The X axis is the data size n, growing from 2^6 to 2^{26}, and the Y axis is the running time in nanoseconds. I used a fitting function to try to fit a line using these data. And the final result is $T(n) = 47.5n$, thus the build operation does have a time complexity of $O(n)$. The coordinates of the result points can be found in appendix 1.

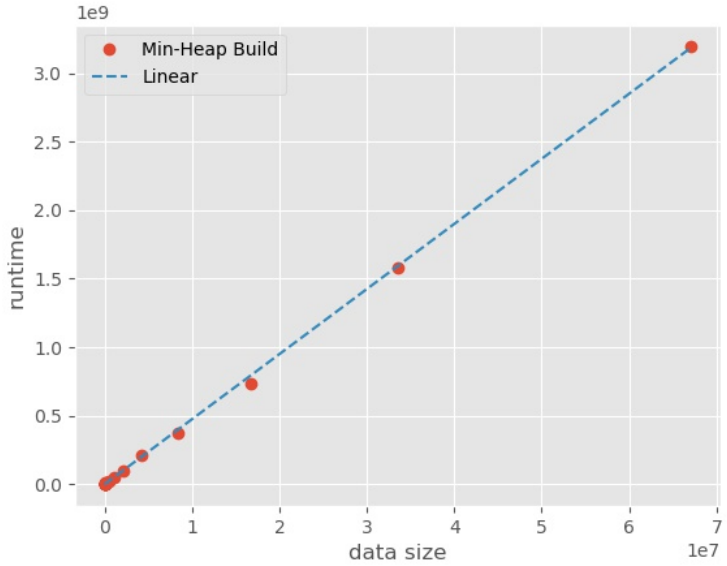

Figure 1: Min-Heap Build

As shown in the figure 2, the red points are the result points. The X axis is the data size n, growing from 2^6 to 2^{26}, and the Y axis is the running time in nanoseconds. The dash curve here is a log function, $17.5 \log n - 40$. It is also clear that the extract-min operation has a $O(\log n)$ time complexity.The coordinates of the result points can also be found in appendix 1.

After running the program for several time, the running time of the peek function is basically constant. The result is [40, 36, 35, 34, 34, 34, 35, 35, 35, 35, 35, 35, 32, 34, 34, 31, 33, 33, 33, 33, 34] for data set size from 2^6 to 2^{26}. It is obvious this is a constant operation, thus there is no corresponding figures.

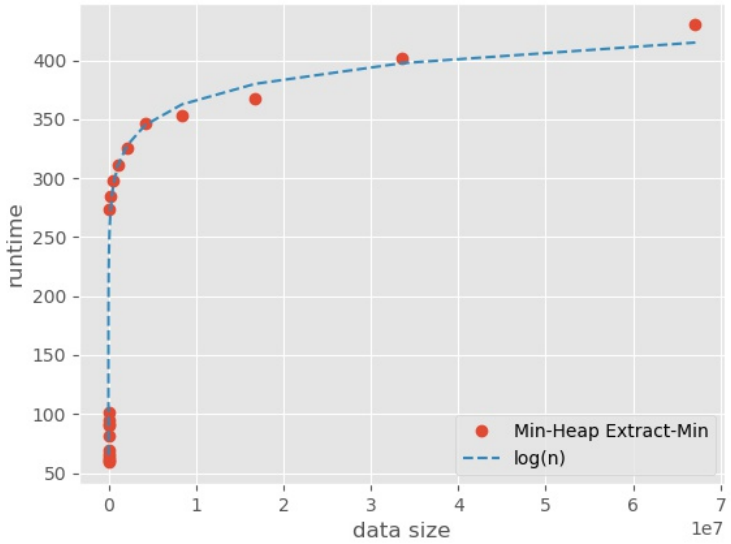

Figure 2: Min-Heap Extract Min

3 Function 2: Search an item with given ID

3.1 Data Structure: Red-Black Tree

A red black tree(RBT) is a self-balancing binary search tree, each node is denoted as either red or black. The main advantage of RBT is that it guarantees searching, inserting, and deleting in $O(\log n)$ time, where n is the number of nodes of the tree, in both average and worst case. Thus, its performance is quite stable. And the AVL tree was not chosen here because of the project context. In a real delivery company warehouse, the insertion and removal operations will be done more frequently than the lookup operation. Since AVL trees provide faster lookup operation, while RBT provides faster insertion and removal operations, I chose to use RBT to store the item information.

3.2 Theoretical Time Complexity Analysis

Though the marking function is the search operation only, we can't do any search without building the tree first. Thus, I developed several functions to do the building and searching, including leftRotate, rightRotate, add, insert, recursiveInsert, find, and search. And each operation has been analysed separately.

3.2.1 leftRotate and rightRotate

These are two helper functions used to ensure that the red black tree follows the red-black properties. They are quite similar, and the only difference is one mainly work on the right child node, while the other mainly work on the left child node.

All statements inside the leftRotate function (line 2-12) are constant, since it only change the pointer of the node, thus this operation run in $O(1)$ time. Similarly, the rightRotate function also run in $O(1)$ time.

```
1  private void leftRotate(Node node) {
2      if (node.parent != null) {
3          if (node.parent.left == node) {
4              node.parent.left = node.right;
5          } else {
6              node.parent.right = node.right;
7          }
8          node.right.parent = node.parent;
9          node.parent = node.right;
10         node.right = node.parent.left;
11         node.right.parent = node;
12         node.parent.left = node;
13     }
```

```
14  }
15  private void rightRotate(Node node) {
16      if (node.parent != null) {
17          if (node.parent.right == node) {
18              node.parent.right = node.left;
19          } else {
20              node.parent.left = node.left;
21          }
22          node.left.parent = node.parent;
23          node.parent = node.left;
24          node.left = node.parent.right;
25          node.left.parent = node;
26          node.parent.right = node;
27      }
28  }
```

3.2.2 add, insert, recursiveInsert, and build

Adding a node to a red black tree needs first adding the node like to a ordinary binary search tree (done by recursiveInsert function), then re-color that node as a red node, and re-schedule the current tree to guarantee it follows the red black properties (done by insert function). Due to these two operations are private, there is an additional public method add to let users add item into the tree. And the build operation is the constructor of the red black tree class calling add function for several times.

Thus, we first need to analyse the time complexity of the recursiveInsert operation. This is the same as the insertion operation in a binary search tree. So the line 2 and line 9 is checking if the node belongs to the left child tree of the root or the right child tree of the root. This takes constant time. Then it check if the left or right child tree is empty in line 3 and 10, if yes, then change the pointers (line 4-5, 11-12), if not, call this recursive method again on this left/right child node.

The worst case is obvious, if the set of nodes is inserted in increasing/decreasing order, the tree will be just like a normal array. Thus the time complexity will be $O(n)$.

We also know that at level d, a full tree will have at most 2^d nodes. Thus the total number of nodes for a full tree is $n = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1$. Then we have $d = \log(n+1) - 1$, i.e. the height of a binary search tree with n nodes can be no less than $\log n$, which means, the recursiveInsert operation should run no less than $\log n$ time. Thus, the average case for recursiveInsert is $O(\log n)$

```
1   private void recursiveInsert(Node root, Node node) {
2       if (root.item.more(node.item)) {
3           if (root.left.item == null) {
4               root.left = node;
5               node.parent = root;
6           } else {
7               recursiveInsert(root.left, node);
8           }
9       } else if (root.item.less(node.item)) {
10          if (root.right.item == null) {
11              root.right = node;
12              node.parent = root;
13          } else {
14              recursiveInsert(root.right, node);
15          }
16      }
17  }
```

Then we need to analyse the time complexity of the insert operation, the first step is doing recursiveInsert operation in line 2-6 which takes $\log n$ time since a red black tree is a balanced tree, except for the first time when the root node is empty, which takes $O(1)$ time. Then it will check the red black properties, from line 8 to 37, it only changes the pointer or color of a node. Such statements all takes constant time, thus the overall time complexity of insertion operation is $O(\log n) + o(1)$, i.e. $O(\log n)$. Since the red black tree is a balanced tree, so both in average and worst case the time complexity of insertion is always $O(\log n)$.

```
1   private void insert(Node node) {
2       if (this.root == null) {
3           this.root = node;
```

```
    } else {
        recursiveInsert(root, node);
    }
    while (!node.item.equals(root.item) && node.parent.isRed()) {
        boolean isLeft = node.parent == node.parent.parent.left;
        Node uncle = isLeft ? node.parent.parent.right : node.parent.parent.left;
        if (uncle.isRed()) {
            node.parent.color = Color.BLACK;
            uncle.color = Color.BLACK;
            node.parent.parent.color = Color.RED;
            node = node.parent.parent;
        } else {
            if (node.item.equals(isLeft ? node.parent.right.item : node.parent.left.item)) {
                node = node.parent;
                if (isLeft) {
                    if (node.item.equals(root.item)) root = node.right;
                    leftRotate(node);
                } else {
                    if (node.item.equals(root.item)) root = node.left;
                    rightRotate(node);
                }
            }
            node.parent.color = Color.BLACK;
            node.parent.parent.color = Color.RED;
            if (isLeft) {
                if (node.parent.parent.item.equals(root.item)) root = node.parent.parent.left;
                rightRotate(node.parent.parent);
            } else {
                if (node.parent.parent.item.equals(root.item)) root = node.parent.parent.right;
                leftRotate(node.parent.parent);
            }
        }
    }
    this.root.color = Color.BLACK;
}
```

The running time of the add operation is the running time of insertion plus the initialisation of a new node, which is a constant operation, thus the time complexity of add is still $O(\log n)$ in both worst and average case. And the build operation (the constructor) here just initialise the root node (constant time) then call the add operation for n times, thus the time complexity is $O(n \log n)$ in both worst and average case.

```
public void add(Item item) {
    insert(new Node(item));
}
public RedBlackTree(Item[] items) {
    this.root = null;
    for (Item i : items) {
        this.add(i);
    }
}
```

3.2.3 find and search

The search operation in a red black tree has no difference with the search in a binary search tree. As proved in the section 3.2.1, the height of a binary search tree is no less than $\log n$, and the red black tree is a balanced tree, thus the height is always $O(\log n)$. So the search operation(the find method below) in a red black tree can be done in $O(\log n)$ in both average and worst case. Thus, the search method here is the time of running find operation plus constant time of checking if the returned node is null or not. So the search method here also takes $O(\log n)$ time.

```
private Node find(Node node, int itemID) {
    if (node.item == null) return null;
    if (itemID == node.item.getId()) {
        return node;
    } else if (itemID < node.item.getId()) {
```

```
        return find(node.left, itemID);
    } else {
        return find(node.right, itemID);
    }
}
public Item search(int itemID) {
    Node node = find(root, itemID);
    if (node != null) {
        return node.item;
    }
    return null;
}
```

3.3 Empirical Time Complexity Analysis

3.3.1 Methodology and Data

The methodology of doing empirical time complexity analysis for the red black tree is the same as the methodology described in section 2.1.1. The data sets are the same as well, the only difference is the key of the node is the item ID rather than the deadline date.

3.3.2 Results

Since the build operation is doing n times insertion operations, I measured the running time of building a red black tree from size of 2^6 to 2^25, which is the X axis in figure 3. The Y axis here is the running time in nanoseconds. The dash curve here is a $nlog$ function, $13.1n \log n$. It is clear that the build operation has a $O(n \log n)$ time complexity. The coordinates of the result points can be found in appendix 2.

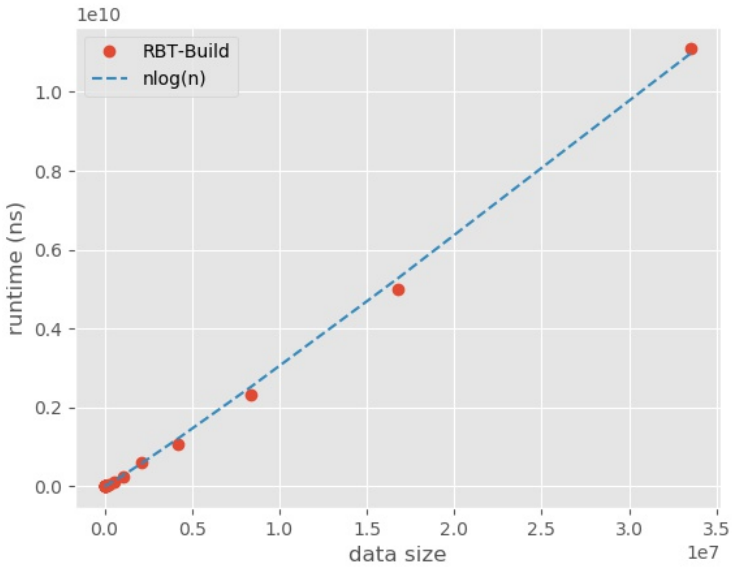

Figure 3: Red Black Tree Build

Since the search takes much less time than build operation, I iterate the search operation for 10000 times and use the average running time to draw scatter chart shown in figure 4. Then I used a log function to fit these dots, $120 \log n - 700$. And it is clear that these dots follows the log trend. Thus the time complexity of insertion in a red black tree is $O(\log n)$. The coordinates of the result points can also be found in appendix 2.

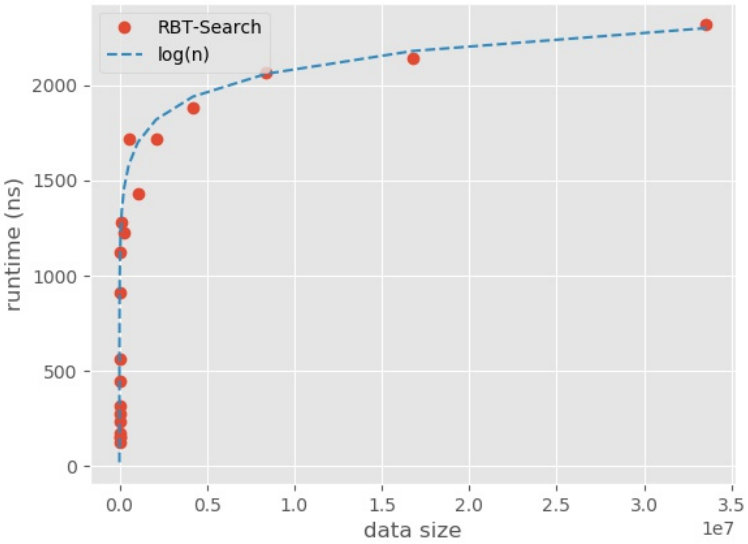

Figure 4: Red Black Tree Search

4 Function 3: Identify the shortest path

4.1 Function: Dijkstra's Algorithm

The Dijkstra's algorithm is a graph search algorithm that can find the shortest path from a start point to an end point for a directed positive weighted graph, which perfectly satisfied this problem conditions. Due to the assumptions I made in section 1.2, all edges in the testing graph are positive weighted. The reason of choosing to use Dijkstra's algorithm is it is faster than the Bellman-Ford algorithm, which is another commonly used graph search algorithm.

4.2 Theoretical Time Complexity Analysis

The pseudo-code for Dijkstra's algorithm is shown below, based on the pseudo-code given by the textbook.

```
1:  function DIJKSTRA(s)                                   ▷ s is the source vertex
2:      queue = new PriorityQueue<Vertex>();
3:      for each vertex v do
4:          v.dist = Integer.MAX_VALUE
5:          queue.add(v)
6:          v.pred = null
7:      end for
8:      s.dist = 0
9:      while !queue.isEmpty() do
10:         u = queue.extractMin();
11:         for each vertex v adjacent to u do
12:             relax(u,v)
13:         end for
14:     end while
15: end function
```

```
1: function RELAX(u, v)
2:     if u.dist + w(u,v) < v.dist then
3:         v.dist = u.dist + w(u,v)
4:         v.pred = u
5:     end if
6: end function
```

Assume the number of vertices is v and the number of edges is e. Line 2 is initialing a priority queue to store all vertices in the graph. Then we have a for loop at line 3 to iterate through all vertices in the graph, and do the enqueue operation at line 5, while line 4 and 6 are constant statements. Thus, we did v times add operations. The line 8 is also a constant statement. The while statement at line 9 iterate through all vertices in the queue, thus the line 10, extractMin operation, will be done once for each vertex, which means in total v times. The for loop at line 11 iterate through all adjacent vertices of u, thus in total this for loop iterates through all edges in the graph, which means it will do the relax operation(decrease-key) for at most e times.

Thus the overall running times should depends on the queue performance, if we use a array, the add and decrease-key operations take $O(1)$ time, while extractMin takes $O(v)$ time, thus overall the running time should be $O(v^2 + e) = O(v^2)$.

4.3 Empirical Time Complexity Analysis

4.3.1 Methodology and Data

The methodology used to do the empirical analysis for Dijkstra's algorithm is slightly different from the previous one. There is no warm-up iterations, while I did 30 iterations on different data size directly.
There are in total 499 data sets, including 100 to 50000 vertices. The size of the data set increases by 100 each time. The graph is connected, positive weighted, directed graph. The number of edges is 3 times of the number of vertices.

4.3.2 Results

There are in total 499 points returned by the program, and is shown in figure 5 as the red line. It is clear that the runtime following the trend of v^2 (blue dash curve), just as proved in the section 4.2. Using regression method, the final results fit the curve $y = 0.0412v^2$.

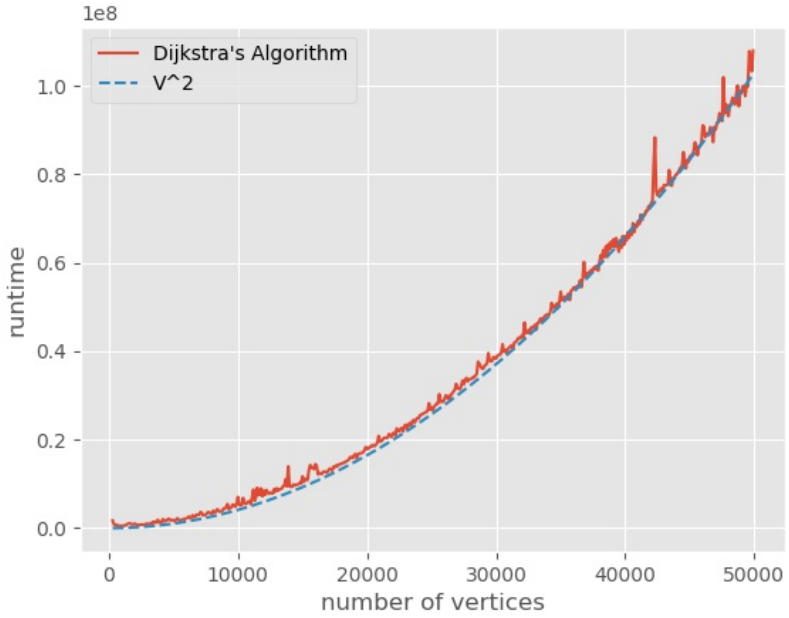

Figure 5: Dijkstra Algorithm

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

Costa, D., Andrzejak, A., Seboek, J., & Lo, D. (2017, April). Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (pp. 389-400).

Appendix 1: Empirical Analysis Data about Min-Heap

Data Size	Min-Heap Build (ns)	Min-Heap Peek (ns)	Min-Heap Extract-Min (ns)
2^6	43225	44	59
2^7	12437	36	62
2^8	17728	35	60
2^9	40308	34	64
2^{10}	75975	34	66
2^{11}	115431	35	69
2^{12}	273364	35	82
2^{13}	589222	35	95
2^{14}	1023455	35	92
2^{15}	1643662	35	90
2^{16}	4562751	35	101
2^{17}	9038009	32	274
2^{18}	13014444	34	285
2^{19}	21366868	34	298
2^{20}	47568255	31	311
2^{21}	101698610	33	325
2^{22}	214777489	33	347
2^{23}	370170664	33	353
2^{24}	732702247	33	368
2^{25}	1583124511	33	402
2^{26}	3191498424	33	430

Appendix 2: Empirical Analysis Data about Red Black Tree

Data Size	RBT Build (ns)	RBT Search (ns)
2^6	359100	136
2^7	200687	121
2^8	238513	160
2^9	115474	186
2^{10}	392493	248
2^{11}	864388	268
2^{12}	1659543	279
2^{13}	1796372	447
2^{14}	3397685	607
2^{15}	7885454	820
2^{16}	13145511	1067
2^{17}	15912477	1195
2^{18}	40986849	1342
2^{19}	99469684	1689
2^{20}	231365966	1467
2^{21}	598768457	1911
2^{22}	1077040431	1819
2^{23}	2314145380	2106
2^{24}	5007431688	2189
2^{25}	11091469905	2411