A Comparison of Data Structures for Dijkstra's Single Source Shortest Path Algorithm

Shane Saunders

November 5, 1999

Abstract

Dijkstra's algorithm computes the shortest paths between a starting vertex and each other vertex in a directed graph. The performance of Dijkstra's algorithm depends on how it is implemented. This mainly relates to the type of data structure used for the frontier set.

This honours project compares the performance of the Fibonacci heap and 2-3 heap implementations of Dijkstra's algorithm. The 2-3 heap is a new data structure invented by T. Takaoka. From the amortized analysis of heap operations, the 2-3 heap and Fibonacci heap implementations of Dijkstra's algorithm have a worst case time complexity of $O(m+n \log n)$. Here n is the number of vertices in the graph and m is the number of edges. If we consider constant factors, worst case analysis gives the number of comparisons, s, as $s = 3m + 1.44n \log_2 n$ for the Fibonacci heap, and $s = 2m + 2n \log_2 n$ for the 2-3 heap.

For random graphs, the average case performance of Dijkstra's algorithm is well within these bounds. To compare the 2-3 heap and Fibonacci heap implementations of Dijkstra's algorithm in detail, we need to consider the average case behaviour. Experimental results for average case processing time and number of comparisons, somewhat reflect the worst case analysis.

Contents

1 Introduction **2**
- 1.1 Dijkstra's Algorithm 3
- 1.2 Importance of the Data Structure Used for the Frontier Set . . . 3

2 Comparing the Implementations of Dijkstra's Algorithm **3**
- 2.1 Comparing the Time Complexity 4
- 2.2 The Binary Heap 4
- 2.3 The Fibonacci Heap 5
- 2.4 The 2-3 Heap 6

3 Analysis of the Different Heaps **9**
- 3.1 Analysis of the Binary Heap 9
- 3.2 Analysis of the 2-3 Heap 10
  - 3.2.1 Amortized Analysis of *insert* 11
  - 3.2.2 Amortized Analysis of *delete_min* 11
  - 3.2.3 Amortized cost for *decrease_key* 12
- 3.3 Analysis of the Fibonacci Heap 18
  - 3.3.1 Amortized Analysis of *insert* 19
  - 3.3.2 Amortized Analysis of *delete_min* 19
  - 3.3.3 Amortized Analysis of *decrease_key* 22
- 3.4 Summary of the Analysis for the Different Heaps. 22

4 Details on Implementing Heaps **23**
- 4.1 Implementing the Binary Heap 23
- 4.2 Implementing the Fibonacci Heap 23
- 4.3 Implementing the 2-3 Heap 24
  - 4.3.1 Representation Used for the 2-3 Heap 24
  - 4.3.2 Choice for Work Space Adjustments 25

5 Generating Random Graphs **27**

6 Expected Results According to the Analysis **29**
- 6.1 The Behaviour of Dijkstra's Algorithm for p=1 30
- 6.2 The behaviour of Dijkstra's algorithm for $p < 1$ 31

7 Results **31**

8 Conclusion **33**

A Program Listings **35**
- A.1 Dijkstra's Algorithm and Random Graph Generation C Program Source 36
- A.2 Binary Heap C Program Source 37
- A.3 Fibonacci Heap C Program Source 38
- A.4 2-3 Heap C Program Source 39

1 Introduction

This honours project compares fast implementations of Dijkstra's single source shortest path algorithm [1]. Dijkstra's algorithm computes the shortest paths from a source vertex to every other vertex in a graph, the so-called single source shortest path problem. The implementations of Dijkstra's algorithm vary in the data structure that is used for the algorithm's frontier set. In this project the Fibonacci heap, 2-3 heap, and binary heap implementations of Dijkstra's algorithm are implemented and compared. For a description of Fibonacci heaps see Fredman and Tarjan's paper [2]. The 2-3 heap is a new data structure invented by Takaoka [3].

Both the Fibonacci heap and 2-3 heap versions of Dijkstra's algorithm are known to have a time complexity of $O(m + n \log n)$, where n is the number of vertices and m is the number of edges in the graph. The binary heap version has a time complexity of $O(m \log n)$. The main aim of this project is to compare the constant factor difference in processing time for Fibonacci heap and 2-3 heap implementations, and relate the experimental results to what is expected from the analysis.

The differences between each implementation of Dijkstra's algorithm, and parameters involved in the comparison, are outlined in Section 2. Section 2.1 introduces the time complexities of each of the implementations of Dijkstra's algorithm. Descriptions of the binary heap, Fibonacci heap, and 2-3 heap, are given in Sections 2.2, 2.3, and 2.4 respectively, to indicate the differences between them.

Section 3 gives an analysis of heap operations for each heap. For the binary heap, worst case analysis of heap operations is used, see Section 3.1. For the 2-3 heap and Fibonacci heap, amortized analysis is given, see Sections 3.2 and 3.3 respectively. Then, in Section 3.4, the analysis of heap operations is combined with the worst case analysis of Dijkstra's algorithm to obtain expressions for the time complexity. This gives the number of comparisons, s, as $s = 3m + 1.44n \log_2 n$ for the Fibonacci heap, and $s = 2m + 2n \log_2 n$ for the 2-3 heap.

This report briefly discusses details involved in implementing each heap, see Section 4. In particular, it may be possible to lower the CPU time involved for 2-3 heap operations, depending on which representation is used to store it, and how the work space is rearranged.

An algorithm for generating the random graphs used by Dijkstra's algorithm was devised and implemented. The uniqueness of this algorithm has not been confirmed. For a description of the algorithm see Section 5. Random graphs produced by this algorithm were used in experiments that compared each implementation of Dijkstra's algorithm.

Section 6 discusses what we expected to be reflected in the experimental results, based on the analysis. Some experimental results are given in Section 7, and related to what is expected from the analysis. Finally, the conclusion is given in Section 8, where we find that the results somewhat reflect the worst case analysis.

The remainder of this introduction continues with Section 1.1 giving a brief introduction to Dijkstra's algorithm. Then, Section 1.2 explains the importance of the data structure that is used for the algorithm's frontier set.

1.1 Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest paths from a starting vertex to all other vertices in a directed graph. In the following description of Dijkstra's algorithm, $OUT(v)$ is defined as the set of all vertices, u, such that there is a directed edge from vertex v to u.

Dijkstra's algorithm maintains three sets for keeping track of vertices: the solution set, S, the frontier set, F, and the set of vertices not in S or F (i.e. unexplored vertices). The set S stores vertices for which the shortest distance has been computed. The set F holds vertices that have an associated *currently best distance* but do not have a determined shortest distance. Any vertex in F is directly connected to some vertex in S. We assume that all vertices in the graph are reachable from the source.

Initially the source vertex, s, is put in S, and the vertices in the OUT set of s are put in F. $^{(*)}$The algorithm proceeds by selecting the vertex v that has the minimum distance among those in F and moves it to S. The shortest distance to v is now known. Then for each vertex, w, that is in $OUT(v)$ but not in S, the following steps are taken:

- A distance, d, is calculated by adding together the shortest distance found for v and the edge length from v to w.
- If w was already in F, then we update w's currently best distance with the minimum of its current value and d.
- Otherwise if w is not in F, then we add w to F and set its currently best distance as d.

This process continues from $(*)$ until F is empty; that is, the shortest distance to all reachable vertices has been computed.

1.2 Importance of the Data Structure Used for the Frontier Set

The process of removing the minimum vertex from F requires a *delete_min* operation on the data structure used for F. Updating the currently best distance requires a *decrease_key* operation and inserting into F requires an *insert* operation. It is important that the data structure used for F supports these operations with a reasonable time complexity. Heaps are well suited for this as they support all of these operations in reasonable time complexity.

The 2-3 heap is a new data structure, developed by Takaoka [3], and has similar time complexity to the Fibonacci heap. The Fibonacci and 2-3 heap both support *delete_min* in $O(\log n)$ amortized time, and *insert* and *decrease_key* in $O(1)$ amortized time. For the binary heap the operations *delete_min*, *decrease_key*, and *insert* all take $O(\log n)$ time.

2 Comparing the Implementations of Dijkstra's Algorithm

Processing time and the number of key comparisons in heap operations, will be used as the main measurement for comparing the three different heap implemen-

tations of Dijkstra's algorithm. To compare the Fibonacci heap and 2-3 heap implementations, this project investigates the constant factors in the computation time and number of machine independent operations. The binary heap implementation of Dijkstra's algorithm is included for a further comparison. Each implementation will be compared over different graph sizes and densities. We will compare the results with those expected from the analysis of each heap and Dijkstra's algorithm. Other parameters, such as graph structure and how the heaps are implemented, can also affect the results.

First, Section 2.1 discusses the time complexity of each implementation of Dijkstra's algorithm. To appreciate the differences between the binary heap, Fibonacci heap, and 2-3 heap, a brief description of each is given. Section 2.2 describes the binary heap, which is the simplest of the three. Then the Fibonacci heap and 2-3 heap are described in Sections 2.3 and 2.4 respectively. Time complexities referred to in these sections relate to the analysis which is given in Section 3.

2.1 Comparing the Time Complexity

The binary heap implementation of Dijkstra's algorithm has a time complexity of $O(m \log n)$, where n is the number of vertices and m is the number of edges in the graph. This was improved on with the Fibonacci heap implementation of Dijkstra's algorithm [2] which has a time complexity of $O(m+n \log n)$. This was shown by Fredman and Tarjan [2] to be the optimal time complexity. The 2-3 heap implementation of Dijkstra's algorithm has a time complexity of $O(m + n \log n)$, the same as the Fibonacci heap implementation.

For high graph densities, the number of edges, m, is comparable to n^2, giving the binary heap implementation a time of $O(n^2 \log n)$ which is worse than the Fibonacci and 2-3 heap implementations' $O(n^2 + n \log n)$ time. However for low graph densities or small graph sizes the binary heap implementation may be faster. This is because the Fibonacci and 2-3 heap implementations have higher computational overhead which should become more significant when the graph size or graph density is small.

2.2 The Binary Heap

A binary heap is a heap ordered tree of a complete binary tree shape. Heap ordered means that the key of any node is not smaller than that of its parent (if any). The binary heap supports heap operations such as *insert*, *min*, *delete*, *delete_min*, *update_key* (that is, *decrease_key* and *increase_key*), and *meld*. For Dijkstra's algorithm, we are mostly interested in the operations *insert*, *delete_min*, and *decrease_key*.

It is possible to implement the binary heap using an array. For a node, i, stored at position i in the array, the left child is stored at $2i$ and the right child (if any) is stored at $2i + 1$. The parent of a node $j > 1$ is stored at j div 2.

Consider a node p whose key has increased, so that it may now be greater than the key of at least one of its children. The rest of the tree is still heap ordered. A process called *sift-up* is used to adjust the sub-tree rooted at p so that the whole tree can be recovered to be heap ordered. Let the minimum child of p be x. If x is smaller than p, we swap p and x. This continues until x is not smaller than p, or the end of the subtree has been reached (p has no children).

The *insert* operation inserts a node, x, at the end of the heap, then adjusts the tree using a *sift-down* process, until the tree is in heap order. The *sift-down* process is similar to the opposite of *sift-up*. We compare x with its parent, p. If x is smaller than its parent, we swap x and p. This continues until x is not smaller than p, or x has became the root of the tree. The insert operation takes at most $O(\log n)$ time.

The *delete* operation removes a node, r, and takes the node at the end of the heap, n, and uses n to replace r. If the key of n is less than the key of r, then a *sift-down* process occurs, otherwise a *sift-up* process occurs. The *delete* operation takes at most $O(\log n)$ time.

The *min* operation finds the minimum node in $O(1)$ time, since the minimum is at the root. The *delete_min* operation deletes the minimum node. This is similar to *delete* except only a *sift-up* process may occur, since the last element is moved to the root. The time complexity of *delete_min* is $O(\log n)$.

The *decrease_key* operation decreases the key of a node x, and uses a *sift-down* process to maintain heap order. The *increase_key* operation increases the key of a node and uses a *sift-up* process to maintain heap order. The time complexity for these operations is $O(\log n)$.

2.3 The Fibonacci Heap

Fredman and Tarjan [2] describe a Fibonacci heap as consisting of a collection of heap-ordered trees. The resulting structure of trees in a Fibonacci heap is described by the way the heap is maintained; there is no explicit constraint on the structure of the trees. Nodes in the heap typically correspond to items which have a key. For this description, where we say "the key of node x", this is equivalent to saying "the key of the item corresponding to node x". We call the number of children of a node its rank. The rank of a tree is defined as the rank of its root node. There is one tree of each rank in the heap.

Since trees are heap-ordered, the root node of a tree gives the node in the tree that has the minimum key. For the collection of trees, which we say are at *root level* in the heap, we typically maintain an array of pointers, where array entry i points to the root node of a tree with rank i. There is an implicit upper bound of $1.44 \log_2 n$ for the number of trees in the heap. In order to locate the minimum node in the heap, we compare the root node of each tree in the heap until the minimum is found.

To describe the structure of a Fibonacci heap, consider what happens when we insert a node into the heap. A node of rank 0 is created and inserted into the heap as a rank 0 tree, possibly performing linking steps. A linking step results when attempting to insert a rank r tree, but there is already a rank r tree in the heap. Since two trees of the same rank are not allowed, they are combined, by comparing the keys of each tree's root node, and linking them such that the root node with the smaller key becomes the parent node of the other tree's root node. This produces a new tree with rank $r+1$, which is then inserted into the heap. If there is already a tree of rank $r+1$, the linking process continues further, until there is an empty position when inserting a tree. This process of insertion is similar to adding 1 to the binary number corresponding to the heap. A zero represents an empty tree position and a one represents a used position. For example, adding 1 to 1111_2 results in four linkings occurring

before the resulting tree can be inserted into the empty position, giving 10000_2. The amortized time complexity for an insert operation is $O(1)$.

When performing a *delete_min* operation, we find the minimum root node and remove it from the heap. The removed root node's child trees are merged back into the root level of the heap, at the array entry corresponding to its rank. Merging can be thought of as reinserting trees into the heap. Each reinsertion can result in several linking steps. This makes the *delete_min* operation the most time consuming step, with an amortized time complexity of $O(\log n)$.

The *decrease_key* operation involves decreasing the key of a node, x, by a given amount, cutting the edge joining x and its parent, p, and merging the tree rooted at x back into the root level of the heap. This causes the rank of p to decrease by 1. For the special case where x is already a root node, we just decrease its key. Because the key is decreased, heap order is maintained in all the nodes below x in the tree. A *decrease_key* operation takes $O(1)$ amortized time.

The *delete* operation is similar to *delete_min* combined with *decrease_key*. After deleting a node x, we merge the sub-trees rooted at each of the child nodes back into the root level of the heap. A *delete* operation has an amortized time bound of $O(\log n)$.

In order to obtain the desired time bounds, there is an additional detail involving the *delete* and *decrease_key* operations. After a root node becomes the child of another node we keep track of whether it has lost a child node. As soon as a node, x, loses two of its children through cuts, we also cut the edge joining x and its parent, p, causing the sub-tree rooted at x to also be merged back into the root level of the heap. This causes the rank of p to be decreased by 1.

This process is called a cascading cut. Several cascading cuts may propagate up a tree from one *decrease_key* or *delete* operation. Cascading cuts are implemented by marking a node after it has lost a child (starting from when it became a non-root node). When cutting a node: if its parent, p, is not marked, then we mark it; otherwise we perform a cascading cut on p.

2.4 The 2-3 Heap

This section gives a brief description of the 2-3 heap invented by Takaoka [3]. The 2-3 heap is somewhat equivalent to the Fibonacci heap, with the same amortized time complexities for each heap operation. However, unlike the Fibonacci heap the 2-3 heap has explicit constraints on the structure of trees in the heap.

First, the structure of the 2-3 heap's heap-ordered trees will be described. The structure of a tree in the 2-3 heap can be described recursively as:

$$\begin{aligned} T(0) &= \text{a single node} \\ T(i) &= T_1(i-1) \bullet \ldots \bullet T_m(i-1) \text{ (where } m \text{ is between 2 and 3)} \end{aligned} \quad (1)$$

The $\bullet$ operator builds a chain of trees joined by edges connecting the root nodes. We call $T(i)$ a tree of dimension i, and the chain of nodes formed from the linking, a trunk of dimension i. A trunk in a 2-3 heap's tree consists of either two or three nodes. In the above description, the subscript i in the notation T_i is used to distinguish each tree in the linking. These trees are of the same

dimension, but their structure can differ since either two or three nodes are allowed in trunks that make up each tree. If we look at the root node of a tree, then T_1 gives the first node in the trunk, called the head node, and T_2 and T_3 give the second and third nodes respectively. A generalisation of the 2-3 heap is the l-r heap which allows m to be between l and r.

As in the Fibonacci heap the 2-3 heap maintains a collection of trees. The dimension of a tree in the 2-3 heap is similar to the rank of a tree in the Fibonacci heap. In Equation 1, for $T(i)$ the linking of $T(i-1)$ trees is called an $(i-1)$th trunk, that is the dimension of the trunk is $i-1$. The dimension of a node is defined as the dimension of the highest dimension trunk it lies on. A root node in a 2-3 heap is the head node for several trunks, starting at a dimension 0 trunk, and moving up to the highest dimension trunk. The Fibonacci heap allows one tree of each rank at root level. In comparison, the 2-3 heap allows up to two trees of each dimension on a main trunk. The maximum dimension tree in the 2-3 heap is bounded by $\log_2 n$.

We can describe the collection of trees in a 2-3 heap as a polynomial of trees. The polynomial for a heap that has up to k main trunks is:

$$P = a_{k-1}T(k-1) + \ldots + a_1T(1) + a_0T(0) \tag{2}$$

The maximum dimension of a tree lying on a main trunk in the heap is $k-1$. Each coefficient, a_i, can be either 0, 1, or 2, corresponding to at most two trees of each dimension. A coefficient a_i represents a linking of trees of dimension i, to form what is called a *main trunk*. The length of a trunk is defined as the number of nodes on the trunk. Main trunks can be either length 0, 1, or 2. Non-main trunks can only be length 2 or 3. All nodes in the trunks, including nodes in main trunks, are maintained in heap order. Main trunks allow for up to two trees of each dimension in the heap.

We now describe the heap operations, beginning with insertion. When inserting a node into a 2-3 heap, a tree of dimension 0 is merged into the heap. For the general description, consider inserting a tree of dimension r. If there are no trees of dimension r in the heap, the new dimension r tree gets inserted. However, if there is already at least one dimension r tree in the heap, we need to compare the keys of the trees' root nodes, and possibly link trees in the heap. Consider the case where there is only one tree of dimension r on the rth main trunk. We link the two trees, extending the corresponding main trunk to include two trees, spending one key comparison to maintain heap order. For the case where there are already two trees of dimension r on the main trunk, we link to create a new trunk of three nodes. The result is a tree of dimension $r+1$ which we remove and merge back into the heap, similar to a carry when doing ternary number addition. The linking to create a 3-node trunk requires either one or two comparisons, depending on the key values of the nodes.

When a tree of dimension $r+1$ is inserted the same process occurs. The overall process of inserting a tree of dimension r into the heap can be viewed as the addition of the ternary number 3^r to the current ternary number corresponding to the heap. In comparison, inserting in the binary heap corresponds to binary addition.

For the *delete_min* operation, we locate the minimum node by comparing the head nodes of each of the main trunks. This is the same as for the Fibonacci heap, considering that the head node of the main trunk is the root of the overall

tree. Suppose the dimension of the minimum node is r. The minimum node is the head node of trunks of dimension $0 \ldots r-1$, and possibly r if there is another dimension r tree on the main trunk. After removing the minimum node, we can view what remains of the child trunks as a separate 2-3 heap. The dimension 0 trunk becomes a main trunk of one or two dimension 0 trees, and the dimension $r-1$ trunk becomes a main trunk of one or two dimension $r-1$ trees. If the main trunk had two nodes, we are left with the main trunk holding one dimension r tree. This resulting collection of 2-3 trees can be melded with the rest of the 2-3 heap.

The meld process is similar to the addition of the two ternary numbers which correspond to the two heaps being merged. To meld, we start at the main trunks holding dimension 0 trees and merge them. We can merge two main trunks of dimension r trees reasonably efficiently. The result is at most a possible carry in the form of a dimension $r+1$ tree, plus one tree left on the main trunk. The merging continues at each dimension, including possible carries from previous merges. When implemented efficiently, a merge requires between zero and two key comparisons.

We view merging as adding three ternary numbers: a, the existing main trunk; b, the added main trunk; and c, a carried tree from the previous. At a merging step, for dimension r trunks:

- If there is a carry, c, then merge c with b to give a sum, s, and a new carry, c.
  - If this results in a 3-node trunk, then s is null, and we have the carry tree, c, which we proceed to merge at dimension $r+1$.
  - Otherwise if c is null, but s is not null, we need to merge a with s, giving a new s and c, before proceeding to dimension $r+1$.

For merging two trunks, each with one or two nodes, determine which main trunk has the head node with the smaller key, spending one comparison. Call the main trunks a and b, where a is the trunk with the smaller head node. Call the head nodes on the trunks a_1 and b_1, with $a_1 < b_1$. The second nodes a_2 and b_2 are possible, but may not exist.

- If a_2 does not exist, then link the trunks as $s = a_1, b_1, b_2$ (b_2 is optional), taking no further comparisons. $c =$ null.

- Else if b_2 does not exist, then spend one further comparison to link the trunks as either $s = a_1, a_2, b_1$ or $s = a_1, b_1, a_2$. $c =$ null.

- Else both a_1 and a_2 exist. This gives the result $s = a_2$ and $c = a_1, b_1, b_2$, taking no further comparisons.

The merging of trees back into the 2-3 heap after a *delete_min* operation is much tidier compared to the equivalent process in a Fibonacci heap. For the Fibonacci heap, the merging of each tree is done separately, instead of as one binary addition of all the trees. We cannot merge the trees back into the Fibonacci heap as one binary addition since there is no guarantee that there is at most one tree of each rank being merged. The entire process for a *delete_min* operation in a 2-3 heap has $O(\log n)$ worst case time.

In order to explain the *decrease_key* operation, it is necessary to describe how a node and its sub-tree are removed from a 2-3 heap. The work space of a node is defined as follows. Suppose the dimension of a node, v, is r, that is, the highest dimension trunk it lies on is the rth. Consider the $(r+1)$th trunk of the head node of v. The work space includes all nodes on rth trunks whose head nodes are on this $(r+1)$th trunk (including the rth trunk of v). With trunk lengths of 2 or 3, the size of the work space can be between 4 and 9 nodes.

A *decrease_key* operation requires removing a tree after the key has been decreased, then merging it back into the heap at main-trunk level. The removal of a tree, $tree(v)$, rooted at node v, located on an ith trunk, proceeds as follows:

- We do not remove the tree if it is the root, that is, the head of a main trunk.
- If the highest trunk that v lies on contains three nodes, (u, v, w), or if v lies on a main trunk, (u, v), then we remove $tree(v)$. This shrinks a non-main trunk to length 2 and a main trunk to length 1.
- Otherwise the highest trunk has just two nodes and we must fill v's position by relocating other nodes in the work space:
  - As long as the work space contains at least five nodes this can be done, spending *at most* one comparison.
  - If the work space has just four nodes, then we remove v and rearrange the remaining three nodes to give a 3-node ith trunk. This requires at most one comparison. The overall result of this whole process can be viewed as the removal of a node on the $(i+1)$th trunk. Thus, the removal process continues at a higher dimension, where nodes in the work space may again be rearranged. Removal can continue several dimensions higher, and will stop once we do not encounter a workspace of size 4.

3 Analysis of the Different Heaps

This section gives the analysis of each heap's operations, examining the difference in amortized cost for each of the heap operations of the Fibonacci heap and the 2-3 heap. We analyse the cost as the number of key comparisons. Time is proportional to the cost. Section 3.1 explains the $O(logn)$ time complexity of operations of the binary heap. In Section 3.2 the amortized cost for the 2-3 heap operations *insert*, *delete_min*, and *decrease_key* is determined. Similar analysis for the Fibonacci heap in Section 3.3. Finally, in Section 3.4 the analysis of heap operations is summarised, and related to the time complexity of Dijkstra's algorithm.

3.1 Analysis of the Binary Heap

Let the depth of the binary tree be d. Assume that every node has two children, except for the leaf nodes at the lowest level. Then, the number of nodes is $n = 2^d$. Therefore the depth of the binary heap's tree is at most $\log_2 n$. The *sift-down* and *sift-up* processes, as described in Section 2.2, sift down or up at

most d levels. This means that the *sift-down* and *sift-up* processes take, in the worst case, $O(\log n)$ time.

As a result, the *insert*, *delete_min*, and *decrease_key* operations all have $O(\log n)$ worst case time.

3.2 Analysis of the 2-3 Heap

The worst case cost for the 2-3 heap implementation of Dijkstra's algorithm is $2m + 2n\log_2 n$. To arrive at this result, amortized cost analysis of the 2-3 heap operations *insert*, *delete_min*, and *decrease_key*, is used.

For the amortized analysis, we use the 'potential' technique. Potential for trunks of varying lengths in the heap is defined in Table 1. As defined in Section 2.4, trunk length is the number of nodes on the trunk. The potential, Φ, can be

trunk length	potential, Φ
1	0
2	1
3	3

Table 1: Definition of the potential of trunks in a 2-3 heap.

thought of as an investment. Creating a trunk of length 3 is an investment, since in the future we can remove a node from this trunk, without having to spend any key comparisons. The potential of a 2-3 heap is the sum of the potentials of all its trunks. The notation s_i is used for the amortized cost of the ith heap operation, and a_i for the actual cost, where cost is in terms of the number of key comparisons. The amortized cost of a heap operation is defined as the number of key comparisons minus the change in potential, that is $s_i = a_i - (\Phi_i - \Phi_{i-1})$. The overall amortized cost, s, is the sum of the amortized costs of individual heap operations:

$$s = \sum_i s_i$$

Similarly for the overall actual cost, a:

$$a = \sum_i a_i$$

If there is a total of N heap operations, and we sum the amortized cost of heap operations then:

$$\begin{aligned} s &= s_1 + s_2 + \ldots + s_N \\ &= \big(a_1 - (\Phi_1 - \Phi_0)\big) + \big(a_2 - (\Phi_2 - \Phi_1)\big) + \ldots + \big(a_N - (\Phi_N - \Phi_{N-1})\big) \\ &= a_1 + a_2 + \ldots + a_n + (\Phi_N - \Phi_0) + \big((\Phi_1 - \Phi_1) + (\Phi_2 - \Phi_2) + \ldots (\Phi_{N-1} - \Phi_{N-1})\big) \\ &= a_1 + a_2 + \ldots + a_n + (\Phi_N - \Phi_0) \\ &= a + (\Phi_N - \Phi_0) \end{aligned} \quad (3)$$

We see that the potential terms in the sum cancel, leaving $\Phi_N - \Phi_0$. It is assumed that the potential at the start and end is zero. For Dijkstra's algorithm this is indeed the case, since we begin with an empty heap and end with an empty heap. This means that we are left with $s = a$, that is, the total amortized time is equal to the total actual time.

3.2.1 Amortized Analysis of *insert*

Consider the general case for merging a tree of dimension i, into a main trunk of dimension i. For this general case, assume the tree is already part of the heap, so we are not increasing the number of nodes in the heap. Merging this single tree, is equivalent to merging the existing dimension i main trunk with a dimension i main trunk of length 1. The potential of the length 1 main trunk being merged is zero.

There are three cases:

- The existing main trunk has no nodes. In this case zero comparisons are spent and the potential remains at zero, giving an amortized cost of zero.
- The existing main trunk has one node. In this case one comparison is spent to make a 2-node trunk, and the potential of the trunk changes from zero to one, giving an amortized cost of $1 - (1 - 0) = 0$.
- The existing main trunk has two nodes. One or two comparisons are spent inserting the node to make a 3-node trunk. The result is at most two comparisons, and the potential changes from 1 to 3, giving at worst $2 - (3 - 1) = 0$ amortized cost. Additionally, there is the cost of merging the newly created 3-node trunk into the $(i+1)$th main trunk, but the cost of this is also zero by the same analysis. Therefore the overall amortized cost is zero, even if we have chain carries.

For an *insert* operation, we insert the new node by merging a tree of dimension 0 onto the dimension 0 main trunk. Initially, we must make the new dimension 0 tree part of the heap. Doing this will not change the potential since the potential of the length 1 main trunk, representing the tree before merging, is zero. Therefore overall an *insert* operation has an amortized cost of zero.

3.2.2 Amortized Analysis of *delete_min*

This section describes the amortized cost for a *delete_min* operation. Consider each of the child trunks of the minimum node; that is, each trunk of which the minimum node is the head. The *delete_min* operation decreases the length of all these child trunks by 1, spending no comparisons. Consider the two cases for the change in length of a trunk:

- A 3-node trunk becomes a 2-node trunk, changing the potential from 3 to 1. Amortized cost is $0 - (1 - 3) = 2$.
- A 2-node trunk becomes a 1-node trunk, changing the potential from one to zero. Amortized cost is $0 - (0 - 1) = 1$.

The result of removing the minimum node shortens at most $\log_2 n$ trunks, with an amortized cost of at most 2 for each trunk shortened, giving at most $2\log_2 n$ amortized cost. Consider melding the collection of shortened trunks back into the heap; the amortized cost of this is zero, so the overall amortized cost of a *delete_min* operation is $2\log_2 n$.

The process of melding trunks back into the heap can be shown to have zero amortized cost. The merging process effectively takes each trunk, which has either one or two nodes in it, and merges it with an existing main trunk in the

heap. If the trunk being merged into the heap has length 1, then the analysis for merging is as described for insertion, see Section 3.2.1. Compared to insertion, the tree being merged is already part of the heap.

Consider a trunk of length 2 being merged back into a main trunk of the heap. For this merging, there are three possible cases of the existing main trunk:

- No existing trunk. Insert with no comparisons and no change in potential. Amortized cost is zero.
- Existing main trunk has one node. Merge, spending one or two comparisons, to give a 3-node trunk. Before merging we had a 1-node trunk and a 2-node trunk, giving a potential of 1. After merging, we have a 3-node trunk, giving us a potential of 3. Amortized cost, considering at most two comparisons, is $2 - (3 - 1) = 0$. The cost of inserting the newly created three node trunk, by the same analysis, is zero.
- Existing main trunk has two nodes. Merge, spending one comparison, to give a 3-node trunk plus a 1-node trunk. Before merging we had two 2-node trunks, giving a potential of 2. After merging, the 3-node trunk plus a 1-node trunk gives a potential of 3. Amortized cost is $1 - (3 - 2) = 0$. The cost of inserting the newly created 3-node trunk into the next main trunk, by the same analysis, is zero.

For every possible case of merging trunks, the amortized cost is zero.

3.2.3 Amortized cost for *decrease_key*

A *decrease_key* operation involves removing the decreased node (and the subtree rooted at it) then merging the node back into the main trunk level of the heap. In order to analyse the *decrease_key* operation, consider the different cases that arise when removing a node from the work space, and the number of comparisons and change in potential for each. If *decrease_key* is applied to the root node of a tree in the 2-3 heap (the first node on the main trunk), then there is no need to remove it, since heap order is still maintained. We do allow the second node on a main trunk to be removed, treating it in the same way as the third node on other trunks.

Let the number of nodes in the work space be w. Assuming the node being removed is not on a main trunk, the work space consists of one $(i+1)$th trunk plus up to three ith trunks off it. Normally w is between 4 and 9, except in the special case when the $(i+1)$th trunk is a main trunk of length 1.

Consider any ith trunk of length 3, $(head, 2nd, 3rd)$. The third node can be removed, shrinking the trunk to length 2. The second node can be removed, with the third node replacing it, shrinking the trunk to length 2. This does not cost any comparison and decreases the potential by 2, since the potential of one ith trunk has decreased from 3 to 1. Therefore removing a node from an ith trunk of length 3 has an amortized cost of 2.

Removing a node from an ith trunk of length 2, affects other trunks in the work space. For this description, label the nodes in the work space according to Figure 1. Nodes are labelled relative to the node being removed, r. The ith trunk that r lies on is (h, r), where h is the head node. Where ith trunks lie immediately above and below trunk (r, h), we use the labelling (a_1, a_2, a_3) and

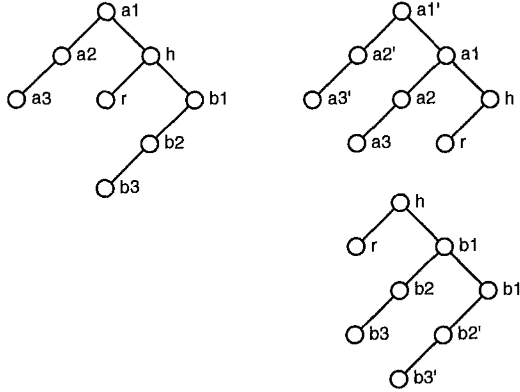

Figure 1: Referring to the work space nodes relative to the node being removed from a 2-node trunk.

(b_1, b_2, b_3) respectively. For a further trunk above or below (r, h), we have either (a'_1, a'_2, a'_3) or (b'_1, b'_2, b'_3). When referring to a trunk with just two nodes, we use a labelling such as, for example, (a_1, a_2). For specific cases, some trunks and nodes are not applicable, since they do not exist.

When removing r from the length 2 ith trunk, we are left with a length 1 ith trunk. Since we do not allow trunks of length 1 we must rearrange the work space. There are several rearrangement possibilities, depending on what trunks and nodes are available in the work space. In certain situations, more than one possibility may be applicable, so we have a choice; that is, an overlap in possibilities can occur. The rearrangement possibilities, named according to the trunks involved, are given below:

bbb – If the trunk (b_1, b_2, b_3) or (b'_1, b'_2, b'_3) exists, we can shorten a b trunk to replace r with one of the b nodes, spending no comparisons. Generally, a 2-3 heap implementation would use one of b_2, b_3, b'_2, or b'_3, which still maintains heap order. The less simple option of using a b_1 or b'_1 node, when it lies at the end of the $(i+1)$th trunk, also exists. It depends on the particular 2-3 heap implementation, which node is used to replace r.

Overall, a length 3 trunk becomes length 2, causing potential to decrease by 2, giving an amortized cost of 2.

bb – If the 2-node trunk (b_1, b_2) exists, then we can relocate (b_1, b_2) to make an ith trunk (h, b_1, b_2), spending no comparisons. The same applies for the trunk (b'_1, b'_2), if it exists.

Overall: we lost a length 2 ith trunk ($\Delta\Phi = -1$); a length 2 ith trunk is now length 3 ($\Delta\Phi = +2$); and the length of the $(i+1)$th trunk may change. The overall change in potential is $+1$ plus the change in potential of the $(i+1)$th trunk.

For a length 3 $(i+1)$th trunk becoming length 2, the overall change in potential is -1, giving an amortized cost of 1. For a length 2 main trunk becoming length 1, the overall change in potential is zero, giving an

amortized cost of zero. We also have the case where $w = 4$, which will be described later, where removal proceeds to a higher dimension. We may treat this as no change in the length of the $(i+1)$th trunk, so the overall amortized cost is -1 plus that for removal at the higher dimension.

aaa – If the 3-node trunk (a_1, a_2, a_3) exists and no b trunk exists, we can 'bend' the work space to give the two ith trunks (a_1, h) and (a_2, a_3), with (a_1, h) above (a_2, a_3). This uses no comparisons. The overall change is that a 3-node trunk is now a 2-node trunk, corresponding to a change in potential of -2, so the amortized cost is 2.

An alternative when the 3-node trunk (a_1, a_2, a_3) exists, is to use a_3 to replace r. This requires 1 comparison and the potential decreases by 2, giving an overall amortized cost of 3. Because the amortized cost for this alternative is more it is not used.

aa – If the 2-node trunk (a_1, a_2) exists, we can relocate h to make the trunk (a_1, a_2, h) or (a_1, h, a_2). The same applies if the trunk (a'_1, a'_2) exists.

This requires one comparison. The change in potential is similar to that for possibility *bb*, being either -1 or 0, giving amortized costs of 2 or 1 respectively. For $w = 4$ the amortized cost is zero plus that of removal at a higher dimension.

We see that when removing from a 2-node trunk the amortized cost is at most 2. This holds true for case $w = 4$, where removal proceeds to higher dimensions, since the cost at a previous dimension would have been either 0 or -1 (a gain of 1). It is interesting to note that only possibility *bbb* requires comparing the keys of nodes.

The current analysis for removing a node has shown that the amortized cost is at most 2. For completeness, the analysis for each work space size is given. Cases can arise where there is a choice of how the workspace is adjusted, and the cost is the same for each possible adjustment. Where there is a choice of which work space adjustment to use, this is left for the particular 2-3 heap implementation to decide; see the implementation described in Section 4.3.2.

- $w = 9$: Figure 2 shows work spaces that can result when removing any node from a work space of size $w = 9$. Only the six black nodes may be removed. If we were removing one of the white nodes, then the work space would be one dimension higher up. There are three trunks of dimension i and one of dimension $i + 1$ in the work space. We spend no comparisons, and the potential of the work space starts at 12 and decreases to 10, giving an amortized cost of 2.

- $w = 8$: Refer to Figure 3. In this case, if we remove a node from one of the two 3-node ith trunks, then the analysis is similar to that for $w = 9$; the amortized cost is 2. If we are removing the black node on a 2-node trunk, we use either possibility *bbb* or *aaa* to rearrange the workspace, depending on the workspace shape. Both possibilities *bbb* and *aaa* have an amortized cost of 2. For all rearrangements, the work space potential changes from 10 to 8, and we spend zero comparisons, giving an amortized cost of 2.

- $w = 7$: Figure 4 shows examples of removing nodes from work spaces of size 7. For this case, there is one 3-node ith trunk and two 2-node ith

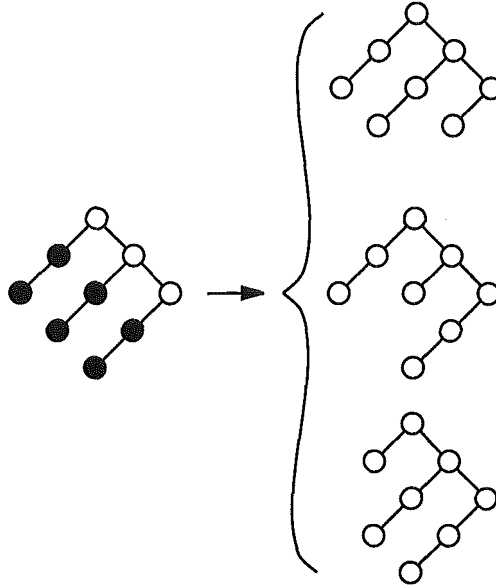

Figure 2: Removing a node from a work space of size 9.

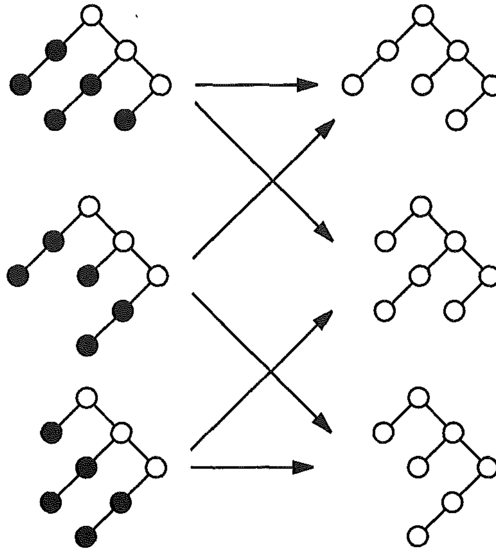

Figure 3: Removing a node from a work space of size 8.

trunks. If we remove a node from the 3-node ith trunk, the amortized cost is 2. If we remove a node from one of the 2-node ith trunks, then possibilities bbb, bb, aaa, and aa could occur. In this case the respective

cost for each possibility is 2, 1, 2, and 2. A case can arise where there is a

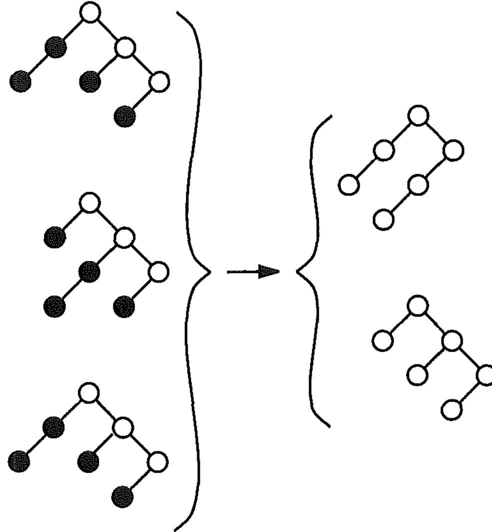

Figure 4: Removing a node from a work space of size 7.

choice between possibilities *bbb* and *aa*, both of which have an amortized cost of 2. The choice of which one to use is left for the particular 2-3 heap implementation. There is also a case where a choice exists between possibilities *aa* and *aaa*, each with an amortized cost of 2.

The potential can change from 8 to 7, spending either 0 or 1 comparisons. Also, the potential can change from 8 to 6, spending 0 comparisons. In any case, the amortized cost is at most 2.

- $w = 6$: There are two possible arrangements of nodes that give a work space of size 6, see Figure 5. The first arrangement has three ith trunks of length 2, with the $(i+1)$th trunk being length 3. For this arrangement, we are only removing from a 2-node trunk, and have the possibilities *bb* and *aa*, with respective amortized costs of 1 and 2. The potential changes from 6 to 5, spending either 0 or 1 comparisons. It would cost less by choosing possibility *bb* over possibility *aa*, for the case where both are applicable.

  The second arrangement has two ith trunks of length 3, with the $(i+1)$th trunk being length 2. The amortized cost of removing a node from either ith trunk is 2. The potential changes from 7 to 5, with no comparisons spent.

  Note that the first arrangement, with three ith trunks, may have an amortized cost of only 1, whereas the second arrangement always has an amortized cost of 2. When removing a node from a workspace of size 7, it may be better to favour producing the arrangement that has three ith trunks. Referring to Figures 1 and 4 , consider when $w = 7$ and node a_3 is missing. There is a choice from the rearrangement possibilities *bbb*

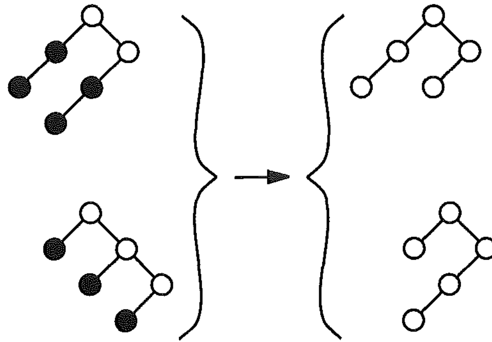

Figure 5: Removing a node from a work space of size 6.

and *aa*. Possibility *bbb* produces three length 2 ith trunks. Possibility *aa* produces two 3-node ith trunks. In the long run it should be better to choose possibility *bbb*.

- $w = 5$: Figure 6 shows the possible results which can occur when removing a node from a work space of size 5. There are two ith trunks, one of length 3, and the other of length 2. Removing from the 3-node trunk has amortized cost 2. When removing from the 2-node trunk, rearrangement possibility *bbb* or *aa* could be applicable. In both cases, the amortized cost is 2. The potential changes from 5 to 3, with no comparisons spent.

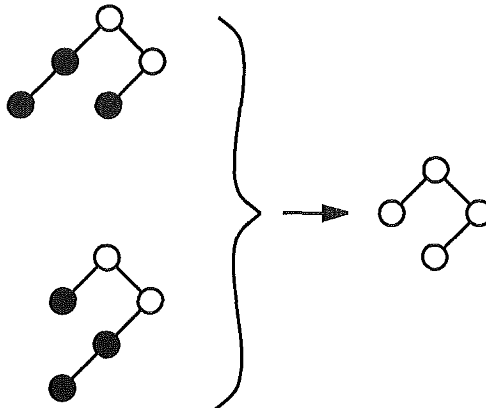

Figure 6: Removing a node from a work space of size 5.

- $w = 4$: For a workspace of size 4, there are two ith trunks of length 2, refer to Figure 7. Rearrangement possibilities *bb* and *aa* are applicable. In both cases, we are left with one ith trunk of length 3. The resulting $(i+1)$th trunk only has length 1, which is not allowed unless it is a main trunk. The overall effect of the rearrangement is equivalent to removing

the second node on the $(i+1)$th trunk. Because of this, removal proceeds in the $(i+1)$th dimension. When removal proceeds to a higher dimension, and it is for a node on a main trunk, we do allow the removal of the second node.

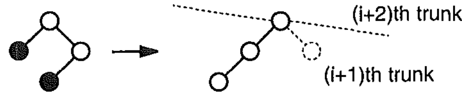

Figure 7: Removing a node from a work space of size 4.

Now consider the number of comparisons and change in potential. Possibility *bb* requires no comparisons to do this, but possibility *aa* requires one comparison. The potential of the work space before removal was 3. The overall effect of the removal is: a length 2 trunk is destroyed ($\Delta\Phi = -1$); a length 2 trunk becomes length 3 ($\Delta\Phi = 2$); and other changes caused by removal at the $(i+1)$th dimension. Overall, we have: zero or one comparisons; an increase in potential of 1; plus the amortized cost for removal at the $(i+1)$th dimension.

The overall amortized cost is equal to -1 or 0 plus the cost for removal at dimension $(i+1)$. Therefore in the worst case, we have 0 plus 2; an amortized cost of 2. Consider what happens when removal propagates several dimensions higher up, with an amortized cost of -1 each time. The new arrangement of the work space in this case can give us an overall gain, rather than a cost.

- **special case:**

  We may have just one ith trunk if the $(i+1)$th trunk is a main trunk of length 1. If the ith trunk has length 3, the amortized cost is 2 as usual. Now consider when the ith trunk (h, r) has length 2. We remove r, decreasing the potential by 1, and then demote h to a lower dimension main trunk, since its rank has decreased. Merging h back into root level has cost 0. The overall amortized cost is 2 if removing from a length 3 ith trunk, and 1 if removing from a length 1 ith trunk. As for the other cases, the amortized cost is at most 2.

The analysis has shown that in all cases the amortized cost of removing a node, and its corresponding subtree, is at most 2. After removing the node it gets merged back into the main trunk level of the heap. Section 3.2.1 showed that the amortized cost of merging is 0. Therefore the amortized cost for a *decrease_key* operation is at most 2.

3.3 Analysis of the Fibonacci Heap

The cost, in terms of the number of key comparisons, for the Fibonacci heap implementation of Dijkstra's algorithm will be at worst $3m + 1.44n\log_2 n$, using amortized analysis. The amortized analysis for the Fibonacci heap uses the 'potential' technique as described in Section 3.2. This analysis is slightly modified

from Fredman and Tarjan's analysis of the Fibonacci heap in [2]. Their analysis uses the term of potential, but by their definition the potential, as the number of trees plus two times the number of marked nodes, is actually a deficit. In [2] the amortized cost is calculated as $s = a + \Delta\Phi$, compared to $s = a - \Delta\Phi$.

For the analysis of the Fibonacci heap in this report, potential is defined as an investment rather than a deficit. This requires a slight modification to the definition of potential given by Fredman and Tarjan [2]. In this report, potential is defined as: the number of nodes in the heap, N_n, minus the number of root nodes, N_r, minus twice the number of marked nodes, N_m. That is $N_n - (2N_m + N_r)$. In this definition marked nodes and root nodes still behave as a deficit.

We now analyse each of the heap operations.

3.3.1 Amortized Analysis of *insert*

First the analysis for merging root nodes is described. Nodes being merged are assumed to have previously been made part of the heap, so that for the description of merging we are not concerned about N_n increasing. Consider a single node with rank r being merged into the root level position r. For the case where the heap does not already contain a node of rank r, the insert increases N_r by 1, spending no comparison. Therefore the overall change in potential is -1, and with no comparisons spent this gives an amortized cost of 1.

Now consider the case where there is already a tree in the heap with rank r. One comparison is spent on linking the two rank r trees to give a rank $r+1$ tree; destroying the root, previously at position r, causes N_r to decrease by 1. The amortized cost of this part, assuming we have not merged the rank $r+1$ tree back into the root level yet, is $1 - 1 = 0$. By the same analysis, the amortized cost to merge the newly constructed rank $r+1$ tree is 1, giving the overall amortized cost of merging as 1. This is because each time a linking step is necessary, the amortized cost is zero, so the sum of the cost of linking steps is zero. Linking steps finish when a free root level position is reached. Merging at the free position has an amortized cost of 1.

For an *insert* operation, the rank of the node being inserted is 0. Making the node become part of the heap increases N_n by 1, with an amortized cost of -1. From the analysis, merging the new rank 0 tree into the heap has an amortized cost of 1. Therefore the overall amortized cost for the *insert* operation is zero.

3.3.2 Amortized Analysis of *delete_min*

For the analysis of *delete_min*, we need to consider the maximum rank possible for the node being deleted. Consider the tree in the Fibonacci heap, rooted at the minimum node. Assume the number of nodes in this tree is n, and assume this tree contains all the nodes in the heap. To determine the maximum rank, consider the trees for each rank which have the minimum number of nodes. The rank of a tree is defined as the rank of its root. In determining these trees, we need to consider that any node in the tree could have lost at most one child, according to the rule for cascading cuts.

Figure 8 shows the trees which have the minimum number of nodes for each rank; call these minimum trees. The minimum tree is constructed as follows:

- The minimum tree for rank 0 is a single node.

- The minimum tree for rank 1 is made by linking two rank 0 trees, to give a tree consisting of two nodes.
- The process for constructing a minimum tree of rank i is to link the roots of a rank $i-1$ and a rank $i-2$ tree, with the root of the rank $i-1$ tree becoming the root for the resulting rank i tree.

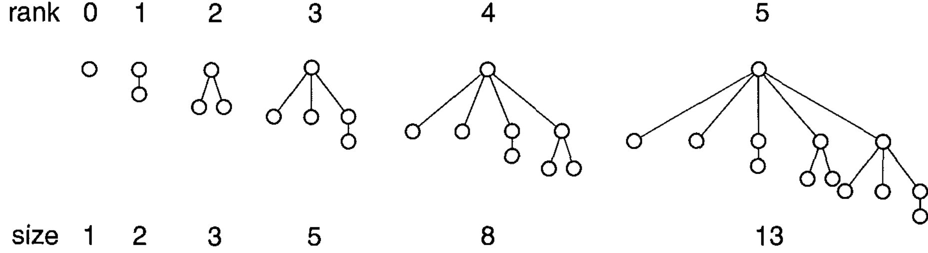

Figure 8: Trees of minimum possible size for a given rank in a Fibonacci heap.

The construction of minimum trees is proved as follows, assuming $i \geq 2$. Normally a rank i tree is constructed by linking two rank $i-1$ trees. If we want to construct the rank i minimum tree, then we would begin by linking two rank $i-1$ minimum trees. Call the two rank $i-1$ trees $T_1(i-1)$ and $T_2(i-1)$, where T_1 is the tree that has the smaller root node, so that $root(T_1)$ becomes the root of the new tree:

$$T(i) = T_1(i-1) \bullet T_2(i-1) \tag{4}$$

The rank i tree resulting from this is not the minimum tree.

Prior to linking $T_1(i-1)$ and $T_2(i-1)$, all the child nodes of $root(T_1(i-1))$ and $root(T_2(i-1))$ are marked, but $root(T_1(i-1))$ and $root(T_2(i-1))$ are not marked. Note that the rank 1 minimum tree with a marked child node is possible to construct by removing the two leaf nodes from a rank 2 tree containing four nodes. Node $root(T_2(i-1))$ is the only unmarked child of $root(T)$, so we may remove a child from $root(T_2(i-1))$.

We can represent the construction of $T_2(i-1)$ as:

$$T_2(i-1) = T_1(i-2) \bullet T_2(i-2) \tag{5}$$

$T_2(i-2)$ is the highest ranking child minimum tree of node $root(T_2(i-1))$. By removing this child from $T_2(i-1)$, we guarantee obtaining the minimum tree for T. This gives

$$T_2(i-1) = T_1(i-2) \tag{6}$$

which means that

$$T(i) = T_1(i-1) \bullet T_1(i-2) \tag{7}$$

Dropping subscripts that were used to distinguish the trees:

$$T(i) = T(i-1) \bullet T(i-2) \tag{8}$$

This shows that the minimum tree of rank i can be constructed from a minimum tree of rank $i-1$ and a minimum tree of rank $i-2$, with $root(T(i-1))$ becoming

the root of the new tree. The resulting minimum tree has all child nodes marked. Proof follows by induction.

For the number of nodes, n, in a minimum tree of rank k:

$$\begin{aligned} n(0) &= 1 \\ n(1) &= 2 \\ n(k) &= n(k-1) + n(k-2) \end{aligned} \tag{9}$$

The sequence of Fibonacci numbers is defined as:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(k+2) &= F(k+1) + F(k) \end{aligned} \tag{10}$$

For a detailed description of the Fibonacci sequence, refer to Knuth [4]. This recurrence relation can be solved to give:

$$F(k) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right) \tag{11}$$

The first term in this equation is more dominant as n tends toward infinity. The golden ratio is defined as:

$$\phi = \frac{1+\sqrt{5}}{2} \tag{12}$$

This approximates the ratio between two consecutive large Fibonacci numbers. It can be shown that the following inequality holds:

$$F(k+2) \geq \phi^k \tag{13}$$

Comparing the definitions for $n(k)$ and $F(k)$, we have:

$$n(0) = 1,\ n(1) = 2,\ n(2) = 3,\ n(3) = 5,\ \ldots$$

compared to:

$$F(0) = 0,\ F(1) = 1,\ F(2) = 1,\ F(3) = 2,\ \ldots$$

We see that $n(k) = F(k+2)$. Then from Equation (13) we get:

$$n(k) \geq \phi^k \tag{14}$$

That is, a node of rank k has at least ϕ^k descendants, including itself. Solving Equation 14 for k, gives an expression for the maximum rank, k, of a tree containing n nodes:

$$k \leq \log_\phi n = \frac{\log_2 n}{\log_2 \phi} = 1.44 \log_2 n \tag{15}$$

Now to analyse *delete_min* consider that the minimum node, which is the root of a tree, has at most $1.44 \log_2 n$ children. Consider the worst case, where none of the child nodes are marked. In this worst case, no marked nodes become

unmarked nodes, so N_m and stays the same, causing no change in the potential. At worst, the cost is from merging $1.44 \log_2 n$ trees back into the root level of the heap. From Section 3.3.1 we know the amortized cost of merging a tree back into the heap is 1. Therefore the overall amortized cost for merging child trunks back into the heap is $1.44 \log_2 n$.

Losing the minimum node from the heap causes no change in potential. One node is lost, decreasing N_n by 1. However, this was a root node, so N_r also decreases by 1, and the overall change in potential is 0. Therefore the overall amortized (and worst case) cost for a *delete_min* operation is $1.44 \log_2 n$.

3.3.3 Amortized Analysis of *decrease_key*

For the analysis of *decrease_key*, we must consider the cost of cascading cuts that occur. Cascading cuts only occur for nodes that are marked. When a cut node that was marked is merged back into root level, it becomes unmarked, and N_m increases, causing potential to increase by 2. The amortized cost for merging such a node is 1. This means that the overall cost from a cascading cut is -1, where the overall effect of a cascading cut is to increase the potential by 1.

The first cut of the *decrease_key* operation, at worst causes an unmarked non-root node to be merged into root level. At worst this merge has an amortized cost of 1; refer to Section 3.3.1. The last cut of the *decrease_key* operation, at worst, causes an unmarked node to become marked. This increases N_m by 2, and thus the potential decreases by 2, so the amortized cost of this is 2.

Totalling the costs of each part in a *decrease_key* operation gives the amortized cost as 3 minus the number of cascading cuts. For the worst case, where there are no cascading cuts, the amortized cost is 3. At worst a *decrease_key* operation has an amortized cost of 3.

3.4 Summary of the Analysis for the Different Heaps.

The summary of the analysis for Fibonacci and 2-3 heap operations is given in Table 2. Dijkstra's algorithm has at most m *decrease_key* operations and n

	Fibonacci Heap	2-3 Heap
insert	0	0
delete_min	$1.44 \log_2 n$	$2 \log_2 n$
decrease_key	3	2

Table 2: Results from amortized analysis of the 2-3 heap and Fibonacci heap operations.

insert and *delete_min* operations. Combining this with the amortized analysis of Fibonacci heap operations, we get an expression for the number of comparisons used by a Fibonacci heap implementation of Dijkstra's algorithm:

$$s = 3m + (0 + 1.44 \log_2 n)n = 3m + 1.44n \log_2 n \qquad (16)$$

Similarly, for the 2-3 heap implementation:

$$s = 2m + (0 + 2 \log_2 n) = 2m + 2n \log_2 n \qquad (17)$$

The binary heap analysis has $O(\log n)$ worst case time for *insert*, *delete_min*, and *decrease_key*. Therefore a binary heap implementation of Dijkstra's algorithm has $O(m \log n + n \log n)$ worst case time. Dropping the second term in the time complexity, since the first term dominates, gives $O(m \log n)$ worst case time.

4 Details on Implementing Heaps

4.1 Implementing the Binary Heap

Binary heaps can be implemented using one dimensional arrays, as used in this experiment. Implementing the binary heap using arrays is easier than implementing using pointers between node structure types.

4.2 Implementing the Fibonacci Heap

The Fibonacci heap is implemented by defining a node structure type similar to the following:

```
/* The structure type for Fibonacci heap nodes. */
typedef struct fheap_node {
    struct fheap_node *parent;
    struct fheap_node *left, *right;
    struct fheap_node *child;
    int rank;
    int marked;
    long key;
    int vertex_no;
} fheap_node_t;
```

Each node in a Fibonacci heap has pointers for its parent node, child node (the head child in a child list), and its left and right siblings. Each node has structure variables for its rank, key, and whether or not it is marked. For this project, the implementation maintains a structure variable `vertex_no` specifically for use with Dijkstra's algorithm so the function for *delete_min* can return the number for the deleted vertex. The set of trees of the Fibonacci heap is maintained using an array of pointers to the root node of each tree.

Several details arise in implementing the Fibonacci heap, which may be used to improve its performance. One such detail is the order in which we merge trees back into the root level of the heap after a *delete_min* or *decrease_key* operation. Consider what happens when we treat merging as adding as binary numbers. This cannot be done easily since some of the nodes to be inserted may have the same rank. Because of this, we have to merge nodes separately, similar to performing separate binary additions, otherwise we need to write a more complicated merging function. Do we merge starting with the highest rank tree to the lowest rank tree, or the other way around? Consider needing to add 1000, 0100, 0010, and 0001 to 1111. If we go from the lowest rank to the highest rank, then the addition proceeds as: 10000, 10010, 10110, 11110. The opposite case proceeds as: 10111, 11011, 11101, 11110. One of these options may produce slightly better CPU time performance.

For this Fibonacci heap implementation, when a *delete_min* operation is performed, we scan the list of root nodes in order to determine which one is the smallest. The array of pointers to the root nodes is fixed at a certain size and we do not want to have to scan the whole array when only the lower portion of it points to root nodes. A pointer to the maximum rank tree in the heap could be maintained in order to avoid this but this is fairly complicated. A simpler option is to maintain a binary value of the root nodes in the heap, which is increased when adding a root node and decreased when removing a root node. The maximum rank can be computed by determining the place of the most significant binary digit (easily done using bit shifts). The cost of maintaining the binary value should be very small.

4.3 Implementing the 2-3 Heap

4.3.1 Representation Used for the 2-3 Heap

The processing time required for the 2-3 heap implementation may be improved depending on how the it is implemented. Unlike the Fibonacci heap, the implementation of the 2-3 heap has several options because of the 2-3 heap's structure. These options are:

- Implement it using the same structure as for Fibonacci heaps. That is, using nodes with parent, child, and sibling pointers. The dimension of a node (similar to rank) and its key value is maintained. This has been implemented for the comparison.

- Implement using a node structure type where each node has an array of pointer pairs to (2nd, 3rd) nodes on each trunk it is the head of. A head pointer that points to the head node of the trunk is maintained, and also the dimension and key value. Implementing using arrays could be simpler and faster than maintaining pointers. If the array of pointer pairs is fixed in size, then this representation requires a fixed $O(\log n)$ space per node. The 2-3 heap then takes $O(n \log n)$ total space, compared to $O(n)$ for the Fibonacci heap. If we use dynamic arrays, the size can be brought back to $O(n)$ but the implementation is more complicated. Also, the time taken for dynamic allocation, such as the `realloc()` C library function, may slow the implementation down. Such an implementation may not work well if `realloc()` often needs to relocate arrays to larger contiguous blocks of memory.

- Instead of using an array, as in the previous point, each node could maintain its own linked list of (2nd, 3rd) pointer pairs. In order to avoid traversing the linked list, the parent pointer could point to the linked list element. An owner pointer could the be used for linked list elements to point to the node that owns the linked list. This introduces some additional complexity in the 2-3 heap's data structure but it could be easier to maintain than the pointer structure used by Fibonacci heaps.

- A new idea as a result of this research is to store node-pairs, with an appropriate structure defined. We can think of a node-pair as being the second and third node of a trunk. A node-pair item contains:

- the keys for the smaller and larger node.
- a pointer to the highest dimension node-pair that the smaller node is the head of. Similarly, a pointer to the highest dimension node-pair that the larger node is the head of.
- left and right sibling pointers to the node-pair one dimension lower and one dimension higher respectively, which share the same head node of the trunk.
- a parent pointer which points to the node-pair that the head node of the trunk belongs to. We also require a Boolean variable to identify which node of the node pair is the head node for the trunk.

After a *delete_min* operation, the merging of child trees into the root level of the 2-3 heap can be implemented very efficiently, similar to adding ternary numbers, allowing all trees to be merged in one step. This is possible since each child tree of the deleted node has a unique dimension. As explained in Section 4.2, the binary equivalent for the Fibonacci heap is not as simple since there could be several child trees with the same dimension.

4.3.2 Choice for Work Space Adjustments

The main aspect of implementing the 2-3 heap is the choice of which way the work space is modified when removing a node; refer to Section 3.2.3. For this projects 2-3 heap implementation, we specify the way in which the work space is modified. Let us use the term *extra node* for the third node on a $(head, 2nd, 3rd)$

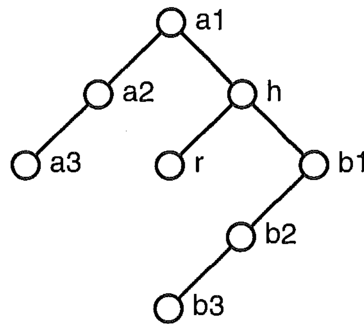

Figure 9: Work space labelling used, relative to the node being removed.

trunk. The second node on a 2-node main trunk is also treated as an extra node. Let the node to be removed be labelled r, and the highest dimension trunk r lies on be the ith. Then if r is an extra node it may be removed, shortening the trunk it lies on, with no further adjustment.

For the case where r is not an extra node, it will be the second node on a non-main trunk. In this case, we have either trunk (h, r) or trunk (h, r, x), where h is the head node of the trunk and x is the extra node, if any. Removing the head node of a main trunk never occurs. For the trunk (h, r, x), after removing r we get (h, x). For the non-main trunk (h, r), we need to consider rearranging

the work space. Figure 9 shows the labelling used for work space nodes relative to r. The labelling used is in Figure 9 is similar to that given in Figure 1 which is explained in Section 3.2.3. However, in this case we only look at other ith trunks immediately above and immediately below r's trunk in the work space; that is, trunks (a_1, a_2, a_3) and (b_1, b_2, b_3). As explained in Section 3.2.3 not all nodes or trunks may be available in any given situation.

As a summary we use the following rules for removing a node, r, and its associated tree:

- If r is an extra node, then remove it without adjustment.
- Else if r is on the trunk (h, r, x), replace r with the extra node, x, to give the ith trunk (h, x).
- Else if node b_3 exists, then replace r with b_3 to give the ith trunk (h, b_3):
  - trim b_3 and replace r with b_3.
- Else if the 2-node trunk (b_1, b_2) exists, then after removing r, rearrange the workspace to get the ith trunk (h, b_1, b_2):
  - trim trunk (b_1, b_2) by recursively removing b_1 in dimension $i + 1$.
  - decrease the dimension of b_1 so that both b_1 and b_2 are dimension i.
  - replace r with (b_1, b_2) to give (h, b_1, b_2).
- Else if node a_3 exists, then after removing r, rearrange the workspace to give the ith trunks (a_1, h) and (a_2, a_3):
  - remove r then twist the work space trunks about a_1. Node h decreases in dimension by 1, node a_2 increases in dimension by 1, and node a_3 retains its current dimension.

  Note that no b trunks would have existed if this case occurs.
- Else if the 2-node trunk (a_1, a_2) exists, then after removing r rearrange the workspace to give the ith trunk (a_1, a_2, h) or (a_1, h, a_2):
  - recursively remove node h at dimension $i + 1$, and decrease its dimension to i.
  - insert h into trunk (a_1, a_2), spending one key comparison to determine where to insert.

  This case can be implemented by comparing the keys of h and a_2 first. If h is smaller, the work space shape gets twisted, swapping h and a_2. Then either h or a_2 can be removed at dimension $i + 1$, and relocated to dimension i to give the trunk (a_1, a_2, h) or (a_1, h, a_2) respectively.
- Else the work space contains just r and h. This case occurs when h is the only node on a main trunk. After removing r, h is demoted to the next lowest dimension main trunk.

5 Generating Random Graphs

To compare the different implementations of Dijkstra's algorithm, we consider comparisons of the processing time taken for directed random graphs of varying edge densities. This comparison requires the graph to be random so that the reported processing time is not for a specific behaviour of Dijkstra's algorithm. We use the assumption that all edges in the graph are reachable from the starting vertex. With this in mind it is necessary to devise a method to create directed random graphs with all vertices reachable from a starting vertex, s.

A simple way to ensure that all vertices are reachable is to enforce the existence of edges from s to every other vertex in the graph. However, that would mean the frontier set of Dijkstra's algorithm is initially occupied with all vertices in the graph except s since the OUT set of s contains all other vertices. In this case the graph is not properly random and a specific behaviour of Dijkstra's algorithm results, which could be reflected in the processing time result. An alternative is to enforce the existence of edges of the form $(i, i+1)$, where we assume $i = 1$ corresponds to vertex s (vertices are labelled from 1 to n), so that there is at least one path through the graph which connects all vertices. This is better than the previous method, but the connecting path still lacks randomness.

We next define a technique which produces a more random connecting path structure in the graph. To enforce that all vertices are reachable, we first generate a random sub-graph consisting of $n-1$ edges. The edges of this sub-graph are part of the final random graph. Note that the edge notation used here is for directed edges; that is, (a, b) represents a directed edge from a to b, where a is the source vertex, and b is the destination vertex. Let V be the set of all vertices in the graph. The sub-graph is determined as follows:

- We begin with a chosen starting vertex, s. At the start, there are and no edges in the sub-graph.
- For each vertex, $b \in (V - \{s\})$, we create a new directed edge, (a, b), which points into b. The source vertex, a, is chosen at random from those vertices in V for which the edge (a, b) would not cause a cycle in the sub-graph constructed so far.

This results in a sub-graph with a random tree-like structure which has all vertices reachable from vertex s. To see this, note that each vertex, excluding s, has exactly one edge pointing into it. Consider what happens if we choose any vertex with an edge pointing to it and trace back through the path of edges. After tracing back an edge there will *always* be another edge to trace back on unless we have arrived at vertex s. This is because every vertex has an edge pointing into it except for the starting vertex. By imposing the restriction that no edge can create a cycle, it is implicitly guaranteed that at least one edge with s as the source vertex will be created. Since there are no cycles in the sub-graph when we trace back through the path of edges, we must eventually arrive at vertex s. Hence all vertices are reachable from vertex s.

Figure 10 gives an example random sub-graph which results from the construction steps shown in Figure 11. The vertices have been arranged in a circle and numbered in the clockwise direction, with the starting vertex, s, numbered zero, at the top. The creation of edges leading to each vertex, $b \in (V - \{s\})$,

then proceeds clockwise. In this case, the edges, in the order they get created, are:

$$(7,1),\ (5,2),\ (0,3),\ (3,4),\ (0,5),\ (5,6),\ (6,7)$$

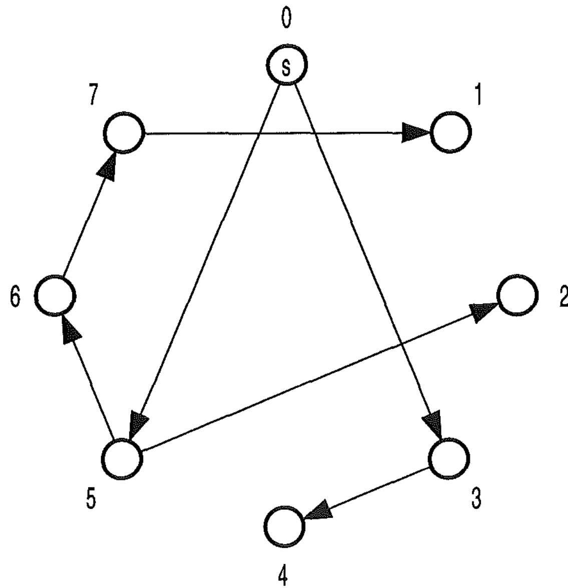

Figure 10: An example random sub-graph.

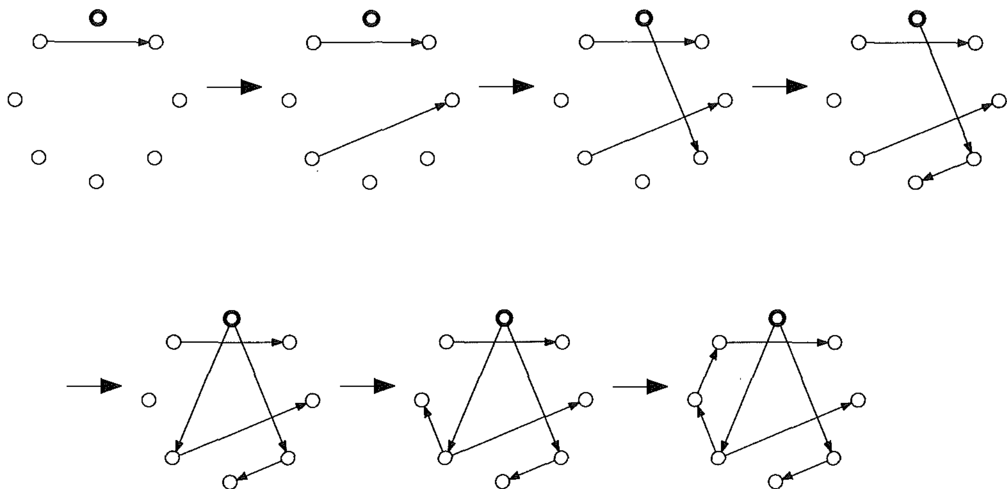

Figure 11: Steps in the construction of the example random sub-graph.

We ensure that there are no loops in the sub-graph by maintaining an array, $c[]$ which keeps track of connected parts of the graph. Vertices are numbered from 1 to n. Array entry $c[i]$ stores the connection number of vertex i. If two edges in the graph are connected then they have the same connection number. That is, the value of $c[i]$ indicates which part of the sub-graph constructed so far, vertex i is connected to. Initially, when there are no edges in the sub-graph, $c[i] = i$ since all vertices are separate and share no connection numbers.

Each time a directed edge, (a, b), is added to the random sub-graph, all vertices with vertex a's connection number, $c[a]$, including a itself, are updated to have vertex b's connection number, $c[b]$. In pseudo code:

```
for all l such that c[l]=c[a] do
    c[l]=c[b]
```

When constructing the random sub-graph, each vertex has at most one edge leading into it. Suppose creating an edge (i, b) is being considered. At this point, vertex b has no edges pointing to it. If $c[i] = c[b]$, then vertex i is connected to vertex b by some path of edges going *from b to i*. The path could not be from i to b since b has no edges pointing to it. Therefore if we were to create the edge (i, b), a cycle would result. Thus, when choosing a source vertex, a, for the random edge (a, b), we only choose from those a's that have $c[a] \neq c[b]$. The process for restricting the choice should be implemented efficiently.

When generating the final random graph, we make sure that it contains all the sub-graph edges, thus ensuring all vertices are reachable from the starting vertex. However, the random graphs produced by this method, may not be perfectly random. When choosing a 'from' vertex for an edge, we restrict those vertices that would cause a cycle to be created in the graph. This may introduce some bias into the structure of the random graph. For the purpose of this project, the graphs produced are random enough. We have not investigated whether this method produces graphs that are properly random. By 'properly random' we mean graphs equivalent to those produced by the following method:

- Keep generating simple random graphs until one is found that has all vertices reachable from vertex s.

A probabilistic method is acceptable for generating graphs that have a moderate to high edge densities, since on average not many graphs would have to be generated before one was found with all vertices reachable from s. However, the performance of this method for graphs with low edge densities is not good. On average, much more graphs would need to be generated since the probability of generating graph with all vertices reachable from s is smaller. This method is probabilistic, so if there were a bound on the worst case time, it is at best exponential in the number of vertices.

6 Expected Results According to the Analysis

We are mainly interested in the number of machine independent operations for comparing the different implementations of Dijkstra's algorithm. When comparing implementations, the CPU time involved will be considered and related to the number of machine independent operations. For notation, s is the number of key comparisons and t is time.

For the Fibonacci heap and 2-3 heap implementations of Dijkstra's algorithm, we have $O(m + n \log n)$ time, giving us the following Equation for describing the number of machine independent operations, s.

$$s = c_0 + c_1 n + c_2 p n^2 + c_3 n \log n$$

Curve fitting can be used to determine these constants from data obtained for S versus n for a fixed value of p. These constants found are compared to the total costs. For the 2-3 heap's amortized cost:

$$2m + 2n \log n = 2pn^2 + 2n \log n \tag{18}$$

For the Fibonacci heap's amortized cost:

$$3m + 1.44n \log n = 3pn^2 + 1.44n \log n \tag{19}$$

Here p is the probability of edge existence.

Equations 18 and 19 are *worst case* amortized costs. The actual behaviour of Dijkstra's algorithm in experiments will give results for an average case behaviour. There may also be tighter bounds on the constants in the above equations. Because of this, experimental results for the number of key comparisons performed by Dijkstra's algorithm are not expected to agree exactly with Equations 18 and 19.

Since the average case behaviour of Dijkstra's algorithm differs significantly from the worst case behaviour, we expect the number of comparisons to be much less than the worst case amortized cost. This report considers results that should be expected for the average case behaviour, but does not do any detailed investigations into the average case behaviour.

First, Section 6.1 considers how the average behaviour of Dijkstra's algorithm differs from worst case behaviour for complete graphs, that is, for $p = 1$. Section 6.2 considers the behaviour of Dijkstra's algorithm for $p < 1$. For a more detailed reference on the expected behaviour of Dijkstra's algorithm for complete graphs, refer to Noshita, Masuda, and Machida [5]. Another reference relating to the expected time complexity of Dijkstra's algorithm is Noshita [6].

6.1 The Behaviour of Dijkstra's Algorithm for p=1

Consider the behaviour of Dijkstra's algorithm for $p = 1$, where each vertex has an edge to every other vertex. Initially the starting vertex, s, is in the solution set, and all other vertices are in the frontier set, since the starting vertex has an edge to every other vertex. The actual number of *decrease_key* operations performed until the frontier set becomes empty is less than n^2, for the following reasons:

- After each *delete_min* operation in Dijkstra's algorithm, the frontier set decreases in size by one, so the maximum number of *decrease_key* operations performed is:

$$(n-2) + (n-3) + \ldots + 2 + 1 = \frac{(n-2)(n-1)}{2}$$

- It is only necessary to perform a *decrease_key* operation if the new distance is shorter than the existing distance to a vertex. Because of this, the number of *decrease_key* operations can be even fewer. The probability of a decrease key changes as Dijkstra's algorithm progresses.

6.2 The behaviour of Dijkstra's algorithm for $p < 1$

The behaviour of Dijkstra's algorithm for specific values of p less than one is more complicated. In such random graphs, the number of edges in the out set of each vertex is not fixed, but we expect the mean number to be about $p \times n$.

If we consider the behaviour of Dijkstra's algorithm for a value such as $p = 0.5$, then the frontier set would initially contain approximately $\frac{n}{2}$ vertices. After each *delete_min* step in Dijkstra's algorithm, one vertex has been removed, but a proportion of vertices previously not in the frontier set or solution set may now have been added to the frontier set. Let us call the set of vertices not in the solution set or frontier set U for 'unexplored'. Each vertex's OUT set contains on average half the other vertices. Using intuition, at each *delete_min* step we would expect half the vertices in U to be moved to F. At the same time, on average *at most* half of the vertices available in F would have a *decrease_key* operation performed on them.

We would expected the number of *decrease_key* operations for $p = 0.5$ to be even less than for $p = 1$. We know this intuitively since the *delete_min* operation is constantly removing items from the frontier set, and for $p = 0.5$ there are fewer vertices in the frontier set on average compared to $p = 1$.

The change in the size of the frontier set for varying values of p can be recorded in order to gain more understanding of the behaviour of Dijkstra's algorithm. We can also keep a count of the number of times each heap operation occurs during a run of Dijkstra's algorithm, or even plot this against time for varying values of p.

7 Results

Results for the number of comparisons involved and the CPU time are shown in Tables 3 and 4. We see from the results that the 2-3 heap implementation of Dijkstra's algorithm runs slightly faster than the Fibonacci heap implementation. The tables also show the percentage fewer comparisons needed using the 2-3 heap compared to the Fibonacci heap. This is used as a measure of the performance increase.

n	Fibonacci	2-3 Heap	% Less
200	2399.6	2283.8	4.8
400	5830.8	5382.6	7.7
600	9686.6	8806.4	9.1
800	13898.8	12434.2	10.5
1000	18055.4	16005.2	11.4
1200	22590.6	19948.2	11.7
1400	27185.2	24009.6	11.7
1600	32050.2	27714.6	13.5

Table 3: Number of Comparisons for Dijkstra's Algorithm, $p = 0.5$

Values in tables were produced by taking the average number of comparisons from running Dijkstra's algorithm on five different random graphs of the given size and edge density. In Table 3, for $p = 0.5$, the performance increase of the

n	Fibonacci	2-3 Heap	% Less
200	1744.6	1541.2	11.7
400	4456.2	3986.2	10.5
600	7594.0	6648.8	12.4
800	10990.8	9333.0	15.1
1000	14571.0	12352.2	15.2
1200	18312.0	15421.6	15.8
1400	22130.4	18805.2	15.0
1600	26022.2	21961.6	15.6

Table 4: Number of Comparisons for Dijkstra's Algorithm, $p = 0.05$

2-3 heap implementation over the Fibonacci heap implementation is greater for larger graph sizes. As graph size, n, increases, the performance increases up to 11 to 13 percent. In Table 4, for $p = 0.05$, the performance increases up to 15 percent.

Preliminary results for the constant factors in computation time have been determined using $p = 1$ for simplicity. For this we use least squares curve fitting on the equation $s = c_0 n^2 + c_1 n \log n$. For simplicity, constant and linear terms are ignored, as these are negligible for large enough values of n. For the data fitted, values of n from 20 to 1700 in steps of 20 have been used. Each data point was taken as the average number of comparisons from five different random graphs. For the Fibonacci heap, we find:

$$s = 0.0014n^2 + 1.78n \log n$$

For the 2-3 heap, we find:

$$s = 0.0007n^2 + 1.67n \log n$$

This result shows that the n^2 component is negligible compared to the $n \log n$ component, except when n is very large. The n^2 component in the worst case amortized analysis arises from the *decrease_key* operations. In the worst case there are $\sim n^2$ *decrease_key* operations in a run of Dijkstra's algorithm, each of which takes $O(1)$ amortized cost. The average case behaviour of *decrease_key* operations appears to be s lower order than n^2, becoming part of the $n \log n$ term.

As an estimate, assume that the $O(n^2)$ term becomes $O(n \log n)$ in the average case. We then convert the worst case amortized cost expression to an equivalent for this average case. Assuming the same constants from worst case analysis, and adding a proportionality constant A, we get $s = 4.44An \log n$ for the Fibonacci heap and $s = 4An \log n$ for the 2-3 heap. The expected performance increase for the 2-3 heap would be $0.44/4.44 \approx 0.10 = 10\%$. This agrees with experimental data. Checking against data for $p = 1.0$ in Table 5, we see that the performance increase approaches approximately 10%.

For CPU time, the 2-3 heap is approximately the same speed as the Fibonacci heap. It is expected that the 2-3 heap implementation could be improved, to lower the CPU time, giving results similar to those for the number of key comparisons.

n	Fibonacci	2-3 Heap	% Less
200	2603.2	2537.8	2.5
400	6252.4	5854.0	6.4
600	10281.8	9467.4	7.9
800	14674.8	13477.2	8.2
1000	19245.2	17273.6	10.2
1200	23976.2	21499.8	10.3
1400	28960.6	25864.6	10.7
1600	33658.6	30207.6	10.3

Table 5: Number of Comparisons for Dijkstra's Algorithm, $p = 1.0$

8 Conclusion

The experimental results indicate that the 2-3 heap requires approximately 10 percent fewer key comparisons than the Fibonacci heap. This agrees with the analysis of the 2-3 heap and Fibonacci heap implementations of Dijkstra's algorithm, which indicated that this should be the case. Overall, the 2-3 heap and Fibonacci heap both have the same amortized time complexities for heap operations. The 2-3 heap has a much simpler, and well defined, structure compared to the Fibonacci heap. This makes the 2-3 heap easier to understand, and maybe simpler to implement if an appropriate representation is used.

Results for CPU time had the 2-3 heap implementation approximately the same speed as the Fibonacci heap implementation. The 2-3 heap used in the experiments was implemented using the same linked list representation as the Fibonacci heap. It is possible to implement the 2-3 heap using a representation that is more appropriate for its structure, such as node-pairs. Such a representation should be easier to maintain, which would lower the overhead involved in the CPU time. As CPU time is typically proportional to the number of key comparisons, it could be possible for an approximate 10 percent speed increase.

Future work in this area may include other forms of l-r heap, such as 2-4 heaps. Further testing may be done, such as the behaviour of the 2-3 heap and Fibonacci heap implementations of Dijkstra's algorithm over specific graph types, for example grid shaped graphs. Fibonacci heap and 2-3 heap implementations of Prim's algorithm, which is similar to Dijkstra's algorithm, may also be tested and compared when using the 2-3 heap or Fibonacci heap.

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[2] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimisation algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[3] T. Takaoka. Theory of 2-3 heaps. In *Proc. COCOON '99*, volume 1627 of *Lecture Notes in Computer Science*, pages 41–50. July 1999.

[4] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[5] Kohei Noshita, Etsuo Masuda, and Hajime Machida. On the expected behaviors of the Dijkstra's shortest path algorithm for complete graphs. *Information Processing Letters*, 7(5):237–243, August 1978.

[6] Kohei Noshita. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms*, 6(3):400–408, September 1985.