

# Shortest Path Algorithms with Integer Edge Costs

Tong-Wook Shinn

7 November, 2002

## **Abstract**

Single source shortest path algorithms are concerned with finding the shortest distances to all vertices in a graph from a single source vertex. Dijkstra (1959) first came up with an  $O(n^2)$  algorithm to solve such a problem, where  $n$  is the number of vertices in the graph. If the given graph has a special property that all edge costs within the graph are integers and these edge costs are bounded by some constant  $c$ , it is possible to improve the underlying data structure of Dijkstra's algorithm to improve the algorithm's time complexity to  $O(m + n \log c)$ .

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Single Source VS All Pairs . . . . .	3
1.2	Improvements on SP Algorithm . . . . .	3
1.3	Third Parameters . . . . .	3
<b>2</b>	<b>Dijkstra's Algorithm and the Frontier Set</b>	<b>4</b>
<b>3</b>	<b>Analysis and Implementation</b>	<b>5</b>
3.1	Integer 2-3 Heap . . . . .	5
3.1.1	2-3 Heap . . . . .	5
3.1.2	Integer 2-3 Heap . . . . .	7
3.1.3	Analysis . . . . .	7
3.1.4	Structure and Implementation . . . . .	8
3.2	Linear Bucket System . . . . .	10
3.2.1	Description . . . . .	10
3.2.2	Analysis . . . . .	10
3.2.3	Structure and Implementation . . . . .	10
3.3	One Level Index . . . . .	11
3.3.1	Description . . . . .	11
3.3.2	Analysis . . . . .	12
3.3.3	Structure and Implementation . . . . .	13
<b>4</b>	<b>Experiments and Results</b>	<b>14</b>
4.1	Number of Comparisons . . . . .	15
4.2	Time Taken . . . . .	15
<b>5</b>	<b>Discussion</b>	<b>17</b>
<b>6</b>	<b>Conclusion</b>	<b>22</b>

# 1 Introduction

Among all types of problems in network optimisation, SP (shortest-path) problems have been some of the most extensively studied. They are commonly encountered in transportation, communication, and production applications, and are often embedded within other types of network optimisation problems (*Optimization Algorithms for Networks and Graphs* 1992).

## 1.1 Single Source VS All Pairs

SP algorithms can be split into two broad categories: single source SP algorithms and all pairs SP algorithms. Single source SP algorithms generate shortest paths from one single source to all other vertices in a graph. The most well-known algorithm for the single source SP problem is Dijkstra's algorithm (1959), which gives an  $O(n^2)$  expected time. An all pairs SP algorithm finds the shortest paths between each pair of vertices. In 1962, Floyd first published an  $O(n^3)$  algorithm for the all pairs SP problem. Only the single source SP algorithm is investigated in this project.

## 1.2 Improvements on SP Algorithm

Since the classical algorithm for single source SP problems was published by Dijkstra, there have been many improvements on the algorithm, both on the analysis and for graphs that belong to particular classes. For example, Fredman & Tarjan (1987) gave an  $O(m + n \log n)$  time algorithm, and if the given graph is planar (the graph can be drawn with no two edges crossing each other) an  $O(n\sqrt{\log n})$  time algorithm can be achieved (Frederickson 1987).

## 1.3 Third Parameters

A third parameter is a variable that describes a certain property in graphs. Abuiaadh & Kingston first introduced the idea of third parameters for the single source SP algorithms (1994). They gave an efficient algorithm for nearly acyclic graphs with  $O(m + n \log t)$  computing time, where  $t$  is the third parameter that represents the number of *delete\_min* operations in the priority queue manipulation. Examples of other algorithms with different third parameters are:

- An algorithm for newly defined acyclic graphs with  $O(m + r \log r)$  computing time, where  $r$  is the number of trigger vertices. Trigger vertices are defined as roots of the resulting trees when a graph is decomposed (Saunders & Takaoka 2001).
- An algorithm with  $O(m + n \log k)$  computing time, where  $k$  represents the maximum cardinality of the strongly connected components in the graph (Takaoka 1998).
- An algorithm for graphs with integer edge costs that has  $O(m + n \log c)$  computing time, where  $c$  is the upper bound on the edge costs (Takaoka 2002).

In this project, the algorithm for graphs with integer edge costs is implemented with various underlying data structures, and the performance of these different data structures are compared against each other and with other existing data structures. Section 2 explains the main concepts of Dijkstra's algorithm and the importance of underlying data structures within the algorithm. Section 3 covers the new data structures that have been implemented for this project, and section 4 contains the experimental results, followed by discussions in section 5, and finally a conclusion is drawn in section 6.

## 2 Dijkstra's Algorithm and the Frontier Set

The single source SP algorithm for graphs with small integer edge costs is based on the original Dijkstra's algorithm. Therefore it is important to understand the main concepts of Dijkstra's algorithm.

In the following description of Dijkstra's algorithm,  $OUT(v)$  is defined as the set of vertices that have directed edges from the vertex  $v$ . Dijkstra's algorithm maintains three sets of vertices: the solution set  $S$ , the frontier set  $F$ , and the set of vertices that are not in either  $S$  nor  $F$ . The set  $S$  stores vertices for which the shortest distances have been computed. The set  $F$  holds vertices that have associated *currently best distances* but do not have determined shortest distances. Any vertex in  $F$  is directly connected to some vertex in  $S$ . All vertices in the graph are assumed to be reachable from the source (Saunders 1999).

Initially the source vertex  $s$ , is put into  $S$ , and the vertices in  $OUT(s)$  are put into  $F$ . Then the algorithm enters a main loop with the following operations:

- Select a vertex  $v$  that has the minimum distance among those in  $F$  and move it to  $S$ . The shortest distance to  $v$  is now known.
- For each vertex  $w$  in  $OUT(v)$  Calculate  $d$  as the sum of the shortest distance found for  $v$  and the length of the edge from  $v$  to  $w$ .
- For each vertex  $w$  in  $OUT(v)$  that are not already in  $F$  or  $S$ , add  $w$  to  $F$ , and set its distance to  $d$
- For each vertex  $w$  in  $OUT(v)$  that are already in  $F$ , compare  $w$ 's current distance with  $d$ , and set its distance to the smaller one.
- Iterate until  $F$  is empty i.e. the shortest distances to all reachable vertices have been computed.

From observing the operations in the main loop of the Dijkstra's algorithm, operations of a priority queue can be identified. The process of removing the vertex that has the minimum distance to the source from  $F$  corresponds to the *delete\_min* operation, updating the current distance in  $F$  corresponds to the *decrease\_key* operation, and adding a vertex to  $F$  corresponds to the *insert* operation of a priority queue. Therefore implementing  $F$  with an appropriate data structure is very important. For example, the difference of implementing  $F$  with a linear array and implementing it with a heap, is comparable to the difference between bubble sort and quicksort.

For graphs with integer edge costs that are bounded by some constant,  $c$ , it is possible to implement special data structures for  $F$  to speed up the single source SP algorithm (Takaoka 2002). This project concentrates on the various data structures that can be used to implement  $F$ .

### 3 Analysis and Implementation

Three special types of data structures for the frontier set  $F$ , in Dijkstra's algorithm have been implemented in this project for graphs with integer edge costs: Integer 2-3 Heap (Takaoka 2002), Linear Bucket System (Takaoka 2002) and One level Index. Detailed description of how these data structures function, what their time complexities are, and how they are implemented, are given in the following sections.

#### 3.1 Integer 2-3 Heap

Integer 2-3 Heap is a new data structure invented by Takaoka (2002). Integer 2-3 Heap is an enhancement over the original 2-3 Heap, which was also invented by Takaoka (1999). Integer 2-3 Heap has been designed specifically to be used to implement  $F$  in Dijkstra's algorithm for graphs with integer edge costs. Since Integer 2-3 Heap is based on the 2-3 Heap, in order to understand how Integer 2-3 Heap works, it is necessary to understand how the original 2-3 Heap works. The following section gives a brief description of the 2-3 Heap. Detailed description of the 2-3 Heap can be found in *Theory of 2-3 Heaps* (Takaoka 1999).

##### 3.1.1 2-3 Heap

The 2-3 Heap is somewhat equivalent to the Fibonacci Heap, with the same amortised time complexities for each heap operations (*delete\_min*, *decrease\_key*, *insert*). Unlike the Fibonacci heap, however, the 2-3 Heap has explicit constraints on the structure of trees in the heap. The structure of a tree in the 2-3 Heap can be described recursively as:

$$\begin{aligned} T(0) &= \text{a single node} \\ T(i) &= T_1(i-1) \bullet \dots \bullet T_m(i-1) \quad (2 \leq m \leq 3) \end{aligned}$$

The  $\bullet$  operator builds a chain of trees joined by connecting the root nodes.  $T(i)$  is said to have a tree dimension of  $i$ , and the chain of nodes formed from the linking is called a trunk of dimension  $i$ . The dimension of a node is defined as the dimension of the highest dimension trunk it lies on. Since  $m$  is between 2 and 3, a trunk in a 2-3 Heap consists of either two or three nodes. In the above notation  $T_i$  is used to distinguish each tree in the linking. The root node of  $T_1$  gives the first node in a trunk, called the head node, and  $T_2$  and  $T_3$  gives the second and third nodes respectively (Saunders 1999).

The 2-3 Heap maintains a collection of trees, and this collection of trees can be described as a polynomial of trees. The polynomial for a 2-3 Heap is:

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

A coefficient  $a_i$  represents a linking of trees of dimension  $i$  to form what is called a *main trunk*. Hence for a heap that has up to  $k$  main trunks, the maximum dimension

of a tree in the heap is  $k - 1$  and this number is bounded by  $\log_2 n$ , where  $n$  is the total number of nodes in the heap. Each coefficient,  $a_i$  can be either 0, 1, or 2, meaning that there are at most two trees of each dimension in the collection of trees in the 2-3 Heap. All nodes in the trunks, including nodes in the main trunks, are maintained in the heap order.

To insert a node into a 2-3 Heap, a tree of dimension 0 is merged into the collection of trees. For the general description, consider inserting a tree of dimension  $r$ . If there are no previous trees of dimension  $r$  in the heap, the new tree gets inserted. If, however, there is already at least one tree of dimension  $r$  in the heap, the keys of the trees' root nodes need to be compared, and possibly the trees need to be linked in the heap. If there is only one previous tree of dimension  $r$ , the two trees are simply linked, spending one key comparison to maintain the heap order. If there are already two trees of dimension  $r$ , the resulting tree after linking 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. In fact, 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.

For the *delete\_min* operation, we locate the minimum node by comparing the head nodes of each of the main trunks. After the minimum node is located and removed, the remains of the trunk can be considered as separate 2-3 Heaps. These resulting 2-3 Heaps can be melded with the original 2-3 Heap, which can be considered as adding two or more ternary numbers, where each ternary number corresponds to a 2-3 Heap that is being merged.

The process of *decrease\_key* is somewhat different to other operations, and it is the key to the great efficiency of the 2-3 Heap. The *decrease\_key* operation of a node  $v$  is performed as follows:

- If  $v$  is the root i.e. the head of the main trunk, no other operations need to be performed.
- If the highest trunk that  $v$  lies on contains three nodes, or if  $v$  lies on a main trunk, then the tree that contains  $v$  needs to be removed and melded back into the original 2-3 Heap.
- If the highest trunk that  $v$  lies on contains just two nodes, other nodes in the work space needs to be reallocated, where the work space of a node is defined as follows: Suppose the dimension of  $v$  is  $r$ . Then, the work space includes all nodes on the  $r$ th trunks whose head nodes are on the  $(r + 1)$ th trunk.
  - As long as the work space contains more than four nodes, this can be done, spending at most one comparison.
  - If the work space has just four nodes,  $v$  is removed and the remaining three nodes are rearranged to give a 3 node trunk. This process can be carried over to higher dimensions, until a workspace of size greater than four is encountered.

### 3.1.2 Integer 2-3 Heap

The 2-3 Heap can be modified in such a way that only integer key values of nodes between 1 and  $c$  are allowed, for a non-negative integer  $c$ . The basic structure and operations of this new Integer 2-3 Heap are much the same as the original 2-3 Heap. The main difference is that in addition to the collection of trees, a set  $S(i)$  for nodes with the key value  $i$  is maintained in a linked list structure. As this project is only concerned with SP algorithms, nodes of the set  $S(i)$  can be thought of as the vertices of the graph, and the key values can be thought of as the distances from the vertices to the source. One of the nodes in  $S(i)$  is called a representative, and others, if any, are called non-representatives. Only the representative nodes can be contained within the collection of trees of the Integer 2-3 Heap. With this property, the three operations of the priority queue are newly defined as follows (Takaoka 2002):

- *insert*: If the key of the inserted node,  $x$ , is  $i$ , then insert  $x$  into the set  $S(i)$ . If  $S(i)$  was empty before insertion,  $x$  becomes the representative of  $S(i)$ , and insert  $x$  into the Integer 2-3 Heap. Otherwise, if  $S(i)$  was not empty,  $x$  simply becomes a non-representative.
- *decrease\_key*: Suppose the key value of a node  $x$  was decreased from  $i$  to  $j$ . Then  $x$  is moved from  $S(i)$  to  $S(j)$ . If  $S(j)$  was empty, remove  $x$  from the Integer 2-3 Heap and reinsert it ( $x$  is still a representative node) to preserve the heap properties. If  $S(j)$  was not empty ( $x$  is no longer a representative node), remove  $x$  from the Integer 2-3 Heap and it becomes a non-representative node. Finally for both cases, if  $S(i)$  is not empty after  $x$  is moved, make a new node its representative node and insert it into the Integer 2-3 Heap.
- *delete\_min*: Scan through the root nodes of the collection of trees in the Integer 2-3 Heap. Suppose the minimum node  $x$  has the key value of  $i$ . After deleting  $x$  from the Integer 2-3 Heap, also remove it from the set  $S(i)$ . If  $S(i)$  is not empty, make a new node its representative node and insert it into the Integer 2-3 Heap.

### 3.1.3 Analysis

The computing time can be analysed by the number of comparisons between the key values, based on amortised analysis (Takaoka 1999). For the original 2-3 Heap, the amortised cost for *delete\_min* is  $O(\log n)$ , for *decrease\_key* it is  $O(1)$ , and for *insert* the amortised cost is  $O(1)$ , as shown by (Takaoka 1999). For the Integer 2-3 Heap, the amortised costs for *decrease\_key* and *insert* are both  $O(1)$ , which are the same as the original 2-3 Heap, but the *delete\_min* operation can be shown to have  $O(\log c)$  amortised cost, where  $c$  is the upper bound in the integer edge costs (Takaoka 2002). This change in time complexity arises because in the Integer 2-3 Heap, only the representative nodes need to be scanned to find the minimum node. Since there can only be at most  $c$  representative nodes, the maximum dimension of a tree in the Integer 2-3 Heap is bounded by  $\log_2 c$ .

Dijkstra's single source SP algorithm has the following properties:

- Number of *decrease\_key* operations performed within the algorithm is proportional to the number of edges,  $m$  in a graph.
- Number of *insert* and *delete\_min* operations performed within the algorithm are both proportional to the number of nodes,  $n$ , in a graph.

With the amortised costs of the three operations and the properties of Dijkstra's algorithm, the time complexity of the single source SP algorithm when the Integer 2-3 Heap is used to implement  $F$ , can be derived:

$$\text{Time complexity} = (O(1))m + (O(1) + O(\log c))n \Rightarrow O(m + n \log c)$$

If  $c < n$ , the  $O(m + n \log c)$  time complexity of the Integer 2-3 Heap is an improvement over the  $O(m + n \log n)$  time complexity of the original 2-3 Heap. If  $c$  is very small, the order function becomes close to linear.

### 3.1.4 Structure and Implementation

The implementation details of the original 2-3 Heap can be found in (Saunders 1999). The Integer 2-3 Heap has been implemented based on the original 2-3 Heap, with the addition of the  $S$  set, and some modifications to the structure of the node and the heap operations. The C code can be found in the appendix.

```
typedef struct ittheap_node {
    struct ittheap_node *prev;
    struct ittheap_node *next;
    struct ittheap_node *parent;
    struct ittheap_node *child;
    struct ittheap_node *left, *right;
    int dim;
    int key;
    int vertex_no;
} ittheap_node_t;
```

The structure of an Integer 2-3 Heap node is shown above. The pointer *\*parent* points to the node's parent, and the pointer *\*child* points to the highest dimension child in a circular doubly linked list of child nodes. The circular doubly linked list of child nodes is maintained using the sibling pointers, *\*left* and *\*right*. The integer *dim* stores the node's dimension, the integer *key* stores the node's key (the distance from the source vertex to this vertex), and finally the integer *vertex\_no* stores the number of the vertex in the graph that this node corresponds to. Variables that have been explained above are from the original 2-3 Heap implementation. Two pointer variables from the node's structure have not been mentioned yet and these are *\*prev* and *\*next*. These two pointers have been added in the Integer 2-3 Heap implementation for the doubly linked lists of the  $S$  set.

```
typedef struct s_list {
    ittheap_node_t *first_node;
    ittheap_node_t *last_node;
} s_list_t;
```

The structure used for  $S$  is shown above. Two pointers that points to Integer 2-3 Heap nodes are implemented in order to maintain a double ended linked list, so that both insertion at the tail of the list and deletion from the head of the list can be performed in  $O(1)$  time.

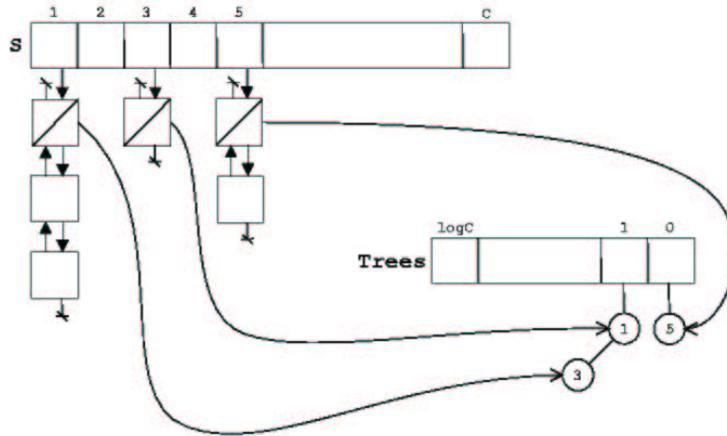


Figure 1: Structure of the Integer 2-3 Heap

```
typedef struct ittheap {
    s_list_t *slist;
    ittheap_node_t **trees;
    ittheap_node_t **nodes;
    int max_nodes, max_trees, n, value;
    long key_comps;
} ittheap_t;
```

Finally shown above, is the structure of the overall Integer 2-3 Heap. The pointer *slist* is used to implement an array of double ended, doubly linked list for the set  $S$ . The pointer *\*\*trees* is for the implementation of the Integer 2-3 Heap trees, while the pointer *\*\*nodes* is used for the implementation of an array of pointers to nodes in the heap so that nodes can be found according to their corresponding vertex number in the graph. *max\_nodes* and *max\_trees* store the maximum number of nodes allowed in the heap and the maximum number of trees allowed in the heap, respectively. *n* stores the current number of nodes in the heap, and *value* stores the current binary value represented by the trees in the heap so that the current maximum rank tree in the heap can be found easily. Lastly, *key\_comps* is used in storing the number of key comparisons, for experimental purposes.

The overall structure of the Integer 2-3 Heap is shown in Figure 1. The figure shows how the set  $S$  is maintained as doubly linked lists. Since only the graphs with integer edge costs bounded by a constant  $c$  is concerned in this project, the length of  $S$  can be set to  $c$  by maintaining a circular structure of  $S$  with the modulo  $c$  operation. The representative nodes are the boxes that are marked with the diagonal lines. The long arrows show which nodes in the  $S$  set correspond to which nodes in the trees of the Integer 2-3 Heap. It is clearly shown in the diagram that there are six nodes altogether within the structure, but only three of these (the representative nodes) are inserted into the collection of trees.

## 3.2 Linear Bucket System

The Linear Bucket System is a simple and well known data structure of a priority queue that can only be used for items with integer key values (Takaoka 2002). The concept behind this data structure is similar to the basic idea of radix sort i.e. no comparisons between the key values are needed to maintain order in the system.

### 3.2.1 Description

The bucket system consists of an array of pointers,  $L$ , that maintains doubly linked lists of items. Let  $L(i)$  be the  $i$ th linked list in  $L$ . Then, if the key of an item  $x$  is  $i$ ,  $x$  appears in  $L(i)$ . Hence in the bucket system, the array positions play the role of key values. The three priority queue operations are described below:

- *insert*: To insert an item  $x$  that has the key value of  $i$ , append  $x$  to the linked list  $L(i)$ .
- *decrease\_key*: To decrease the key of  $x$  from  $i$  to  $j$ , remove  $x$  from the linked list  $L(i)$ , and append it to  $L(j)$ .
- *delete\_min*: To find the item with the minimum key value, the array  $L$  is linearly scanned from the previous position of the minimum key value, until a non empty list  $L(i)$  is encountered. The first item of  $L(i)$  is the item with the minimum key value.

The structure of a bucket system can be thought of as just the  $S$  list part of the Integer 2-3 Heap. The length of  $L$  can be set to  $c$ , similar to the  $S$  list of the Integer 2-3 Heap.

### 3.2.2 Analysis

Clearly, the time complexity for *insert* and *decrease\_key* operations are both  $O(1)$ . The time for *delete\_min* operation depends on the interval between the previous position of the minimum item and the next position of the minimum item. Since this interval is proportional to the length of the array  $L$ ,  $c$ , the following time complexity can be derived if the Linear Bucket System was used in implementing  $F$  of the Dijkstra's algorithm:

$$\text{Time complexity} = (O(1))m + (O(1) + O(c))n \Rightarrow O(m + nc)$$

Since  $\log c \leq c$ , the Linear Bucket System is less efficient than the Integer 2-3 Heap. When  $c$  is very small, however, the difference between  $\log c$  and  $c$  is also very small. The merit of the Linear Bucket System lies in that it is very simple to implement, and yet for small  $c$ , its time complexity is comparable to other more complicated data structures.

### 3.2.3 Structure and Implementation

Figure 2 shows the structure of the Linear Bucket System. The figure shows that  $L(3)$  contains two items with key values of  $c + 3$ , and the current item with the minimum key value is in  $L(6)$ . This illustrates the circular property of the array  $L$ . The structure of an item is shown below:

```

typedef struct bucket_node {
    struct bucket_node *prev;
    struct bucket_node *next;
    int vertex_no;
    int key;
} bucket_node_t;

```

The structure is much simpler than the Integer 2-3 Heap. The two pointers *\*prev* and *\*next* are used to maintain the doubly linked list, while the integer *vertex\_no* stores the number of the vertex in the graph that this item corresponds to, and the integer *key* stores the item's key (the distance from the source vertex to this vertex).

```

typedef struct bucket {
    bucket_node_t *first_node;
    bucket_node_t *last_node;
} bucket_t;

```

The structure of the array of buckets, *L*, is shown above. It is exactly the same as the implementation of *S* in the Integer 2-3 Heap.

```

typedef struct l_buckets {
    bucket_t *buckets;
    bucket_node_t **nodes;
    int max_nodes;
    int n;
    int key_comps;
    int min_pos;
} l_buckets_t;

```

The overall structure of the Linear Bucket System is also quite similar to the Integer 2-3 Heap. There is an array of buckets instead of the *S* set, and the pointers *\*\*nodes*, and the variables *max\_nodes*, *n*, and *key\_comps* are all used in the same way as the Integer 2-3 Heap. There is a new variable called *min\_pos*, however, which is used to store the position in the array of buckets that corresponds to the node with minimum key value. The actual implementation of the Linear Bucket System in C code can be found in the appendix.

### 3.3 One Level Index

One Level Index is a data structure based on the Linear Bucket System. It improves the Linear Bucket System by speeding up the linear scanning process with an *index*, so that the next item with the minimum value key can be found more quickly.

#### 3.3.1 Description

In addition to the array of buckets *L*, of the Linear Bucket System, an index *I*, which is an array of integers, is also kept. *L* is divided into *k* intervals of equal length, such that the length of *I* is also *k*. *I(i)* stores the number of items contained in the *i*th interval of *L*. Then, the process of linear scanning can be sped up by referring to *I*. If *I(i)* is zero, the entire *i*th interval can be skipped. Modifications to the three priority queue operations are described below:

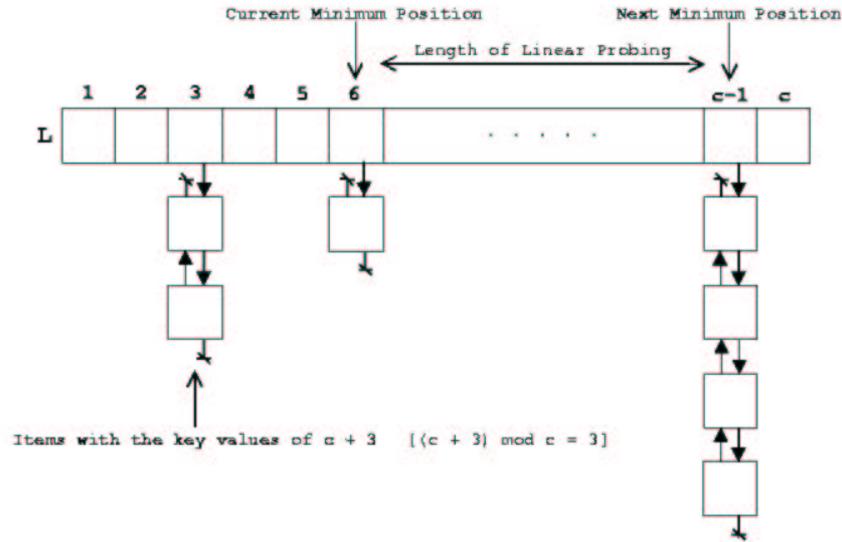


Figure 2: Structure of the Linear Bucket System

- *insert*: To insert an item  $x$  that has the key value of  $i$ , append  $x$  to the linked list  $L(i)$ . If  $i$  belongs to the  $m$ th interval, increment  $I(m)$  by one.
- *decrease\_key*: To decrease the key of  $x$  from  $i$  to  $j$ , remove  $x$  from the linked list  $L(i)$ , and append it to  $L(j)$ . Decrement and increment the corresponding index values by one.
- *delete\_min*: To find the item with the minimum key value, the array  $L$  is linearly scanned from the previous position of the minimum, until a non empty bucket is encountered. If a non empty bucket has not been encountered until the end of the current interval,  $m$  is reached, search the index from  $I(m + 1)$  until a non zero interval,  $j$ , is found. Then, continue the process of linearly scanning the array  $L$  from the start of the  $j$ th interval.

### 3.3.2 Analysis

The time complexity for *insert* and *decrease\_key* operations are both still  $O(1)$ , since finding the corresponding interval and updating the index can both be performed in  $O(1)$  time. The time for *delete\_min* operation depends on the size of the interval. It can be shown that the resulting time complexity is optimal when  $L$  is divided into  $\sqrt{c}$  number of intervals, where the length of each interval is also  $\sqrt{c}$ . The proof is given below:

Since the total length of linear scanning is proportional to the sum of the size of the interval and the length of the index  $I$ , following equation can be derived, where  $y$  is the

time taken for linear scanning, and  $x$  is the number of intervals (length of the index  $I$ ).

$$y = k\left(\frac{c}{x} + x\right) \text{ (for some integer } k\text{)}$$

To solve for  $x$  when  $y$  is minimum,

$$\begin{aligned} \frac{dy}{dx} &= -\frac{kc}{x^2} + k \\ -\frac{kc}{x^2} + k &= 0 \\ k &= \frac{kc}{x^2} \\ x^2 &= c \\ x &= \sqrt{c} \end{aligned}$$

as required, and this gives

$$y = 2k(\sqrt{c})$$

Hence the minimum time complexity for the *delete\_min* operation is  $O(\sqrt{C})$ . Therefore the following time complexity can be derived if the One Level Index data structure is used in implementing  $F$  of the Dijkstra's algorithm:

$$\text{Time complexity} = (O(1))m + (O(1) + O(\sqrt{c}))n \Rightarrow O(m + n\sqrt{c})$$

Since  $\log c \leq \sqrt{c} \leq c$ , the One Level Index is less efficient than the Integer 2-3 Heap, but more efficient compared to the Linear Bucket System.

### 3.3.3 Structure and Implementation

```
typedef struct mbucket_node {
    struct mbucket_node *prev;
    struct mbucket_node *next;
    int vertex_no;
    int key;
} mbucket_node_t;

typedef struct mbucket {
    mbucket_node_t *first_node;
    mbucket_node_t *last_node;
} mbucket_t;
```

The structure for the nodes and the array of buckets are exactly the same as the Linear Bucket System. There are a few additions to the overall structure of One Level Index, however, which is shown below:

```
typedef struct ml_buckets {
    mbucket_t *buckets;
    mbucket_node_t **nodes;
    int *index;
    int sqrtC;
```

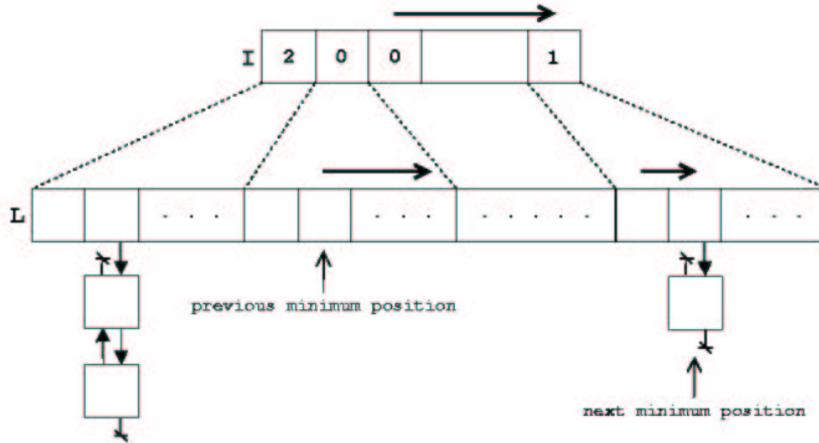


Figure 3: Structure of One Level Index

```

int max_nodes;
int n;
int key_comps;
int min_pos;
} ml_buckets_t;

```

Two more variables are implemented in addition to the overall structure of the Linear Bucket System. The pointer *index*, as the name suggests, is used in implementing the index *I*, and the variable *sqrtC* stores the rounded integer value of  $\sqrt{c}$  to be used in calculations for the maintenance of the One Level Index priority queue. The complete implementation of the One Level Index in C code can be found in the appendix.

Figure 3 shows how One Level Index is implemented. The dotted lines show the region that is covered by the corresponding index. The values 2 and 1 in *I* show that there are two items and one item in the corresponding buckets in *L*, respectively. The bold arrows show the process of linearly scanning *L* and *I* to find the next minimum position. It is shown that the process of searching for the next minimum position is improved by skipping through many empty buckets in *L* with the use of the index *I*.

## 4 Experiments and Results

The three new data structures (Integer 2-3 Heap, Linear Bucket System and One Level Index) were compared against each other and against the original 2-3 Heap by the number of comparisons and/or the time taken to solve the single source SP problems for random graphs. The three different data structures were used in implementing the frontier set, *F*, of the Dijkstra's algorithm, but the Dijkstra's algorithm itself was not changed. The source code of the Dijkstra's algorithm used in this project and the source

$n$	$c = 10$		$c = 100$	
	2-3 Heap	Integer 2-3 Heap	2-3 Heap	Integer 2-3 Heap
1000	13183	2065	13169	4489
2000	29486	4063	29449	8338
3000	47471	6389	47058	12153
4000	65919	8378	65215	15494
5000	84937	10588	84239	18866
6000	104268	12607	103872	22691
7000	124004	14689	123129	26327
8000	144466	16782	144122	29679
9000	165741	19547	164338	33659
10000	186447	21465	184969	36897

Table 1: Number of comparisons for  $c = 10$  and  $c = 100$

code of the program that was used to actually run the experiments are both shown in the appendix.

Random graphs for the experiments were generated using the random graph generator implemented by (Saunders 1999). The source code can be found at <http://www.cosc.canterbury.ac.nz/~tad/alg/graphs/graphs.html>. This random graph generator allows the number of nodes in the graph and the average number of vertices coming out of a node in the graph to be specified by the user. The average number of vertices coming out of a node has been set to 10 for all experiments that have been performed in this project.

#### 4.1 Number of Comparisons

Counting the number of comparisons is not applicable to the Linear Bucket System and the One Level Index, because these two data structures use radix rather than comparison to order numbers. Hence only the Integer 2-3 Heap has been compared against the original 2-3 Heap for this experiment. In counting the number of comparisons, only comparisons used for the maintenance within the two priority queues were counted, and any comparisons required by the Dijkstra's algorithm itself (comparing the initial distance with the new distance after the vertex with the minimum key value has been removed from  $F$ ) were not counted.

Tables 1, 2 and 3 show the number of comparisons taken to solve the single source SP problem with Dijkstra's algorithm for the two different implementations of  $F$ .  $c$  is the upper bound in the integer edge costs of the graphs, and  $n$  is the number of vertices in the graphs. The experiments were performed ten times for each value of  $c$  and  $n$ , and the average value was recorded. Figure 4 is a graph of the number of comparisons with varying values of  $c$ , with  $n = 10000$ , taken from the three tables.

#### 4.2 Time Taken

Time taken for Dijkstra's algorithm to run on graphs with the three new underlying data structures and also the original 2-3 Heap has been measured for this experi-

$n$	$c = 1000$		$c = 10000$	
	2-3 Heap	Integer 2-3 Heap	2-3 Heap	Integer 2-3 Heap
1000	12846	9030	12665	12112
2000	28743	16253	28464	25846
3000	45756	23131	45057	39305
4000	63480	29545	62596	52530
5000	81832	35175	80668	65303
6000	100629	40460	99055	76817
7000	119566	45855	117864	88478
8000	140152	51211	137619	100330
9000	159699	56587	156580	111014
10000	180214	61897	177120	120394

Table 2: Number of comparisons for  $c = 1000$  and  $c = 10000$

$n$	$c = 100000$		$c = 1000000$	
	2-3 Heap	Integer 2-3 Heap	2-3 Heap	Integer 2-3 Heap
1000	12667	12606	12732	12725
2000	28320	28075	28350	28307
3000	44937	44384	44903	44844
4000	62392	61233	62445	62280
5000	80351	78261	80358	80088
6000	98943	96236	98876	98473
7000	117354	112973	117153	116728
8000	136949	130773	136850	136289
9000	155702	149110	155564	154842
10000	175932	166479	176036	174976

Table 3: Number of comparisons for  $c = 100000$  and  $c = 1000000$

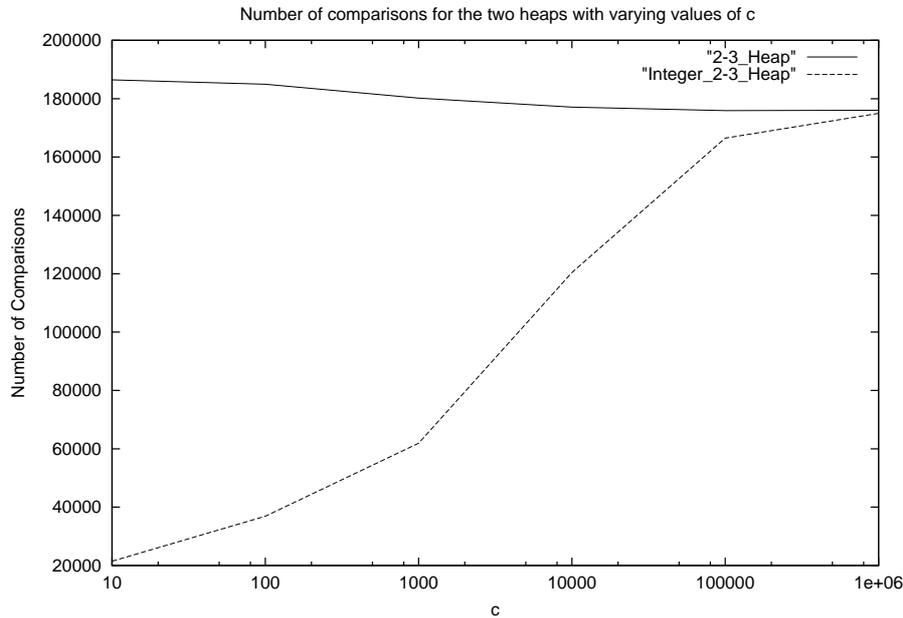


Figure 4: Graph of number of comparisons against different values of  $c$

ment. The time measuring function, implemented by (Saunders 1999), can be found at <http://www.cosc.canterbury.ac.nz/~tad/alg/timing/timing.html>. Times were measured in milliseconds, and the figures shown in the tables from Table 4 to Table 10 are the average of ten different time measurements. Figure 5 shows the graph of the times taken for the four different data structures, with  $n = 10000$ . This experiment was undertaken on a computer with an Athlon 1600 processor and 512 megabytes of RAM.

## 5 Discussion

From the tables and the graph of the number of comparisons, it is clearly shown that the Integer 2-3 Heap performs exceptionally well compared to the original 2-3 Heap for small values of  $c$ . Figure 4 shows the increase in the number of comparisons (decrease in performance) of the Integer 2-3 Heap as the  $c$  value increases. The scale of the x-axis in the graph is logarithmic (base 10), but the increase in the number of comparisons of the Integer 2-3 Heap seems to be linear, which agrees with the previously derived time complexity of  $O(m + n \log c)$  for solving SP problems. Number of comparisons counted for the original 2-3 Heap is almost constant, as the performance of this data structure is not dependent on the value of  $c$ . It can be predicted that as  $c$  is increased further, the performance of the Integer 2-3 Heap will further decrease, but the performance of the original 2-3 Heap will remain constant.

The tables and the graph of the time measurements show a similar trend to the data collected for the number of comparisons. Times measured for the original 2-3 Heap remains constant throughout all values of  $c$ . The three new data structures, however, performs better than the original 2-3 Heap for smaller values of  $c$ , but rapidly degrades

$c = 10$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	0	0	0	0
2000	4	0	2	6
3000	12	8	8	8
4000	16	16	6	14
5000	24	18	16	18
6000	32	24	20	18
7000	36	28	24	24
8000	44	38	28	28
9000	56	40	36	36
10000	66	44	44	44

Table 4: Time taken for  $c = 10$

$c = 100$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	2	2	2	2
2000	8	4	4	6
3000	14	10	10	12
4000	20	16	14	14
5000	26	24	18	22
6000	40	30	24	24
7000	48	42	30	26
8000	58	42	40	42
9000	68	56	42	42
10000	82	62	50	52

Table 5: Time taken for  $c = 100$

$c = 1000$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	2	4	0	0
2000	8	4	6	6
3000	14	10	10	10
4000	18	20	16	12
5000	28	24	20	22
6000	36	32	22	26
7000	50	40	32	32
8000	62	48	38	40
9000	66	54	44	46
10000	80	62	54	52

Table 6: Time taken for  $c = 1000$

$c = 10000$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	2	2	2	0
2000	8	6	2	2
3000	10	10	8	10
4000	20	20	14	14
5000	32	24	16	20
6000	36	36	28	30
7000	42	38	28	28
8000	58	42	42	40
9000	64	48	42	42
10000	76	52	48	50

Table 7: Time taken for  $c = 10000$

$c = 100000$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	2	4	10	10
2000	8	12	12	12
3000	16	18	20	20
4000	20	24	24	30
5000	26	40	32	34
6000	32	46	36	40
7000	42	52	48	44
8000	56	76	48	52
9000	70	82	62	54
10000	76	98	66	62

Table 8: Time taken for  $c = 100000$

$c = 1000000$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	4	26	82	46
2000	10	38	94	70
3000	14	48	122	84
4000	18	60	118	98
5000	28	72	142	102
6000	36	82	150	104
7000	46	100	154	124
8000	56	110	158	130
9000	64	118	176	142
10000	78	138	166	146

Table 9: Time taken for  $c = 1000000$

$c = 10000000$				
$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
1000	0	34	670	146
2000	10	60	692	264
3000	12	98	782	338
4000	16	122	842	404
5000	30	154	938	470
6000	40	188	948	524
7000	42	224	1008	576
8000	54	252	1096	620
9000	68	270	1044	624
10000	80	298	1078	654

Table 10: Time taken for  $c = 10000000$

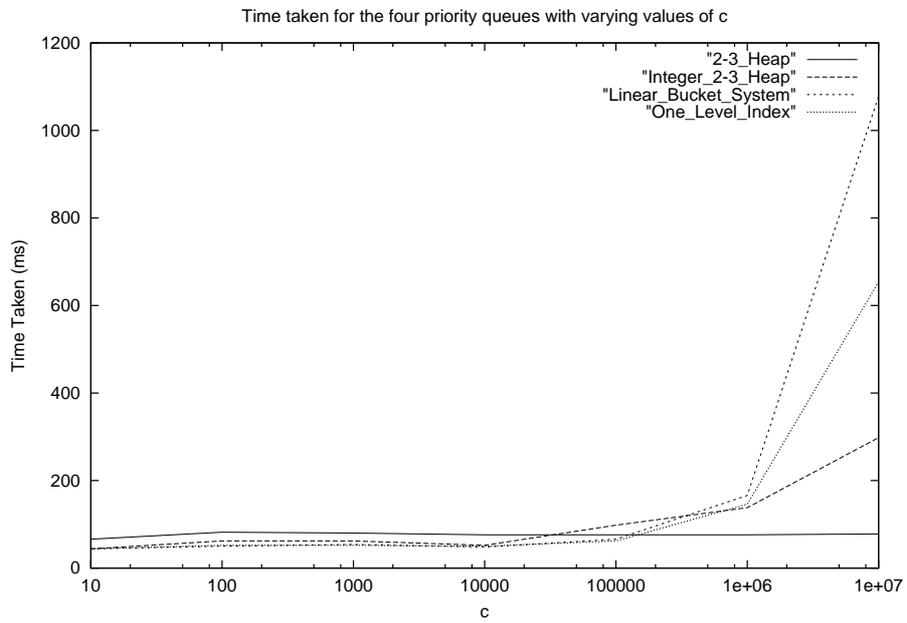


Figure 5: Graph of time taken against different values of  $c$

$n$	2-3 Heap	Integer 2-3 Heap	Bucket	Index
10000	64	110	169	136
20000	185	251	255	223
30000	335	417	369	337
40000	485	593	485	458
50000	629	746	574	555
60000	771	922	672	656
70000	957	1088	777	754
80000	1108	1266	893	869
90000	1310	1496	1040	1016
100000	1447	1629	1119	1092
110000	1625	1827	1235	1206
120000	1824	2029	1350	1335
130000	2007	2229	1479	1462
140000	2197	2429	1598	1592
150000	2388	2612	1733	1731
160000	2556	2824	1850	1852
170000	2771	3042	1992	1986
180000	3041	3321	2213	2211

Table 11: Time taken for  $c = 1000000$  with large values of  $n$

in performance as  $c$  value is increased. The scale of the x-axis is again logarithmic, and the difference between  $c$ ,  $\sqrt{c}$ , and  $\log c$  is clearly shown in Figure 5. Time taken for the Linear Bucket System, which has the overall time complexity of  $O(m + nc)$  increases the most rapidly, followed by the One Level Index, then the Integer 2-3 Heap. The times measured for the Integer 2-3 Heap shows that this data structure is not as efficient in time as the number of comparisons showed. This is because in order to maintain such a complex priority queue, very large number of pointer operations are needed, which slows down the operations by a significant amount.

To confirm that the properties of these data structures hold for larger values of  $n$  (graphs with more vertices), the upper bound in integer edge cost was set to  $c = 1000000$ , and an experiment was undertaken with very large values of  $n$ . In this experiment, an interesting observation was made. Figure 6 shows the graph of the measured times for  $n \leq 60000$ . Since the time complexities of the three new data structures are all dependent on the value of  $c$ , changes in the efficiency of the data structures were not expected no matter what value  $n$  was. Figure 6, however, shows that the Linear Bucket System and the One Level Index actually perform better than the Integer 2-3 Heap and the original 2-3 Heap as  $n$  is increased. This strange behaviour of the Linear Bucket System and the One Level Index can be explained as follows: The reason that the performance of these two structures are dependent on  $c$  is that the length of linear scanning is proportional to the value of  $c$ . When  $n$  becomes large, however, most of the buckets in the array  $L$  will contain items. Therefore the length of linear scanning is greatly reduced as  $n$  is increased.

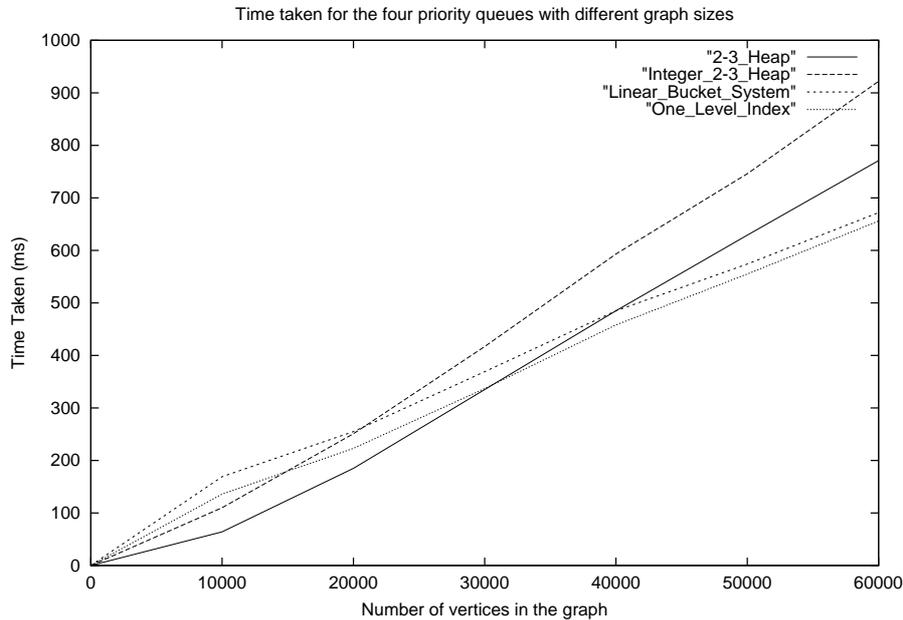


Figure 6: Graph of time taken against different number of vertices

## 6 Conclusion

Given a graph with integer edge costs that are bounded by some constant  $c$ , it is possible to implement a special data structure of a priority queue for the frontier set,  $F$ , of the Dijkstra's algorithm to solve the single source SP problem faster. Three new data structures have been implemented in this project: Integer 2-3 Heap, Linear Bucket System, and One Level Index. Theoretically, Integer 2-3 Heap is the most efficient data structure, followed by One Level Index, then the Linear Bucket System. Experiments show, however, that this order of efficiency does not always hold, and in some cases the Linear Bucket System and the One Level Index performs better than the Integer 2-3 Heap. Linear Bucket System and the One Level Index are both very much simpler to implement than the Integer 2-3 Heap, and also these two priority queues do not need to perform any comparisons to maintain order.

Future work in this area may include the analysis and implementation of new data structures, possibly starting from multi-level indexes. One goal would be to implement a data structure that can solve the SP problem in  $O(m + n \log c)$  time complexity, which does not require any comparisons to maintain the order. Also investigations on other possible third parameters and applications of these data structures to all pairs SP problems may be possible.

## References

- Abuaiadh, D. & Kingston, J. H. (1994), Are fibonacci heaps optimal?, in 'ISAAC'94'.  
\*citeseer.nj.nec.com/abuaiadh94are.html
- Dijkstra, E. (1959), 'A Note on Two Problems in Connexion with Graphs', *Numerische Mathematik* **1**, 269–271.
- Floyd, R. W. (1962), 'Algorithm 97: Shortest path', *Communications of the ACM* **5**(6), 345.
- Frederickson, G. N. (1987), 'Fast algorithms for shortest paths in planar graphs, with applications', *Journal of the SIAM* **16**(6), 1004–1022.
- Fredman, M. L. & Tarjan, R. E. (1987), 'Fibonacci heaps and their uses in improved network optimization algorithms', *Journal of the ACM (JACM)* **34**(3), 596–615.
- Optimization Algorithms for Networks and Graphs* (1992), Marcel Dekker, Inc.
- Saunders, S. (1999), 'A Comparison of Data Structures for Dijkstra's Single Source Shortest Path Algorithm', *Honours Project, Department of Computer Science, University of Canterbury* .
- Saunders, S. & Takaoka, T. (2001), Improved shortest path algorithms for nearly acyclic graphs, in 'CATS 2001'.
- Takaoka, T. (1998), 'Shortest path algorithms for nearly acyclic directed graphs', *Theoretical Computer Science* **1**(203), 143–150.
- Takaoka, T. (1999), 'Theory of 2-3 Heaps', *Technical report, Department of Computer Science, University of Canterbury* .
- Takaoka, T. (2002), 'Shortest paths with limited integer costs', *Technical report, Department of Computer Science, University of Canterbury* .

## Appendix

### Integer 2-3 Heap

```
#include <stdlib.h>
#include <math.h>
#if ITTHEAP_DUMP
#include <stdio.h>
#endif
#include "ittheap.h"
#include "../dijkstra/edgeCost.h"

/** Prototypes of functions only visible within this file. */

void itth_meld(ittheap_t *h, ittheap_node_t *tree_list);
int imerge(ittheap_node_t **a, ittheap_node_t **b);
void iremove_node(ittheap_t *h, ittheap_node_t *cut_node);
void itrिम_xnode(ittheap_node_t *x);
void iadd_child(ittheap_node_t *p, ittheap_node_t *c);
void ireplace_node(ittheap_node_t *old, ittheap_node_t *new);
void iswap_trunks(ittheap_node_t *tr_low,
                 ittheap_node_t *tr_high);

/** Definitions of functions visible outside of this file. */

/* creates and returns a pointer to a 2-3 heap.
 * Argument max_nodes specifies the maximum number of nodes
 * the heap can contain. */
ittheap_t *itth_alloc(int max_nodes)
{
    ittheap_t *h;

    /* Create the heap. */
    h = malloc(sizeof(ittheap_t));

    /* The maximum number of nodes
     * and the maximum number of trees allowed. */
    h->max_nodes = max_nodes;

    /* h->max_trees = 0.5 + log(max_nodes+1)/log(2.0); */
    h->max_trees = 0.5 + log(C+1)/log(2.0);

    /* Allocate space for array S. */
    h->slist = calloc(C + 1, sizeof(s_list_t));

    /* Allocate space for an array of pointers to trees,
     * and nodes in the heap.
     * calloc() initialises all array entries to zero,
     * that is, NULL pointers. */
    h->trees = calloc(h->max_trees, sizeof(ittheap_node_t *));
}
```

```

    h->nodes = calloc(max_nodes, sizeof(ittheap_node_t *));

    /* We begin with no nodes in the heap. */
    h->n = 0;

    /* The value of the heap helps to keep track
     * of the maximum rank
     * as nodes are inserted and deleted. */
    h->value = 0;

    /* For experimental purposes,
     * we keep a count of the number of key comparisons. */
    h->key_comps = 0;

    return h;
}

/* destroys the heap pointed to by h,
 * freeing up any space that was used by it. */
void itth_free(ittheap_t *h)
{
    int i;

    for(i = 0; i < h->max_nodes; i++) {
        free(h->nodes[i]);
    }
    free(h->nodes);
    free(h->trees);
    free(h->slist);
    free(h);
}

/* creates and inserts new a node representing vertex_no with
 * key k into the heap pointed to by h. */
void itth_insert(ittheap_t *h, int vertex_no, int k)
{
    ittheap_node_t *new;
    int index;

    /* printf("itth insert started: %d %d\n", vertex_no, k); */

    /* Create an initialise the new node.
     * The parent pointer will be set to
     * NULL by itth_meld(). */
    new = malloc(sizeof(ittheap_node_t));
    new->prev = NULL;
    new->next = NULL;
    new->child = NULL;
    new->left = new->right = NULL;
    new->dim = 0;

```

```

new->vertex_no = vertex_no;
new->key = k;

/* Maintain a pointer to vertex_no's new node in the heap. */
h->nodes[vertex_no] = new;

/* Now insert it into the S list as well. */
index = k % (C + 1);

if ((h->slist[index]).first_node == NULL) {
    (h->slist[index]).first_node = new;
    (h->slist[index]).last_node = new;

    /* Meld the new node into the heap
       only if it is the only node with vertex_no = index. */
    itth_meld(h, new);
}
else {
    ((h->slist[index]).last_node)->next = new;
    new->prev = (h->slist[index]).last_node;
    (h->slist[index]).last_node = new;
}

/* Update the heap's node count. */
h->n++;
}

/* deletes the minimum node from the heap pointed to by h
 * and returns its vertex number. */
int itth_delete_min(ittheap_t *h)
{
    ittheap_node_t *min_node, *child, *next;
    long k, k2;
    int r, v, vertex_no;
    int index;
    int i;

    /* First we determine the maximum rank tree in the heap. */
    v = h->value;
    r = -1;
    while(v) {
        v = v >> 1;
        r++;
    };

    /* Now locate the root node with the smallest key,
     * scanning from the maximum rank root position,
     * down to rank 0 root position. */
    min_node = h->trees[r];
    k = min_node->key;

```

```

while(r > 0) {
    r--;
    next = h->trees[r];
    if(next) {
        if((k2 = next->key) < k) {
            k = k2;
            min_node = next;
        }
        h->key_comps++;
    }
}

/* We remove the minimum node from the heap
   but keep a pointer to it. */
r = min_node->dim;
h->trees[r] = NULL;
h->value -= (1 << r);
h->n--;

/* A nodes child pointer always points to the child
   with the highest rank,
   so child->right is the smallest rank.
   For melding the linked list starting at child->right,
   we terminate the circular link with a NULL pointer. */
child = min_node->child;
if(child) {
    next = child->right;
    next->left = child->right = NULL;
    itth_meld(h, next);
}

index = min_node->key % (C + 1);

/* Remove it from the S list as well. */
(h->slist[index]).first_node =
    ((h->slist[index]).first_node)->next;

if ((h->slist[index]).first_node == NULL) {
    (h->slist[index]).last_node = NULL;
}
else {
    ((h->slist[index]).first_node)->prev = NULL;
    itth_meld(h, h->slist[index].first_node);
}

/* Record the vertex no to return. */
vertex_no = min_node->vertex_no;

/* Delete the old minimum node. */

```

```

    h->nodes[vertex_no] = NULL;
    free(min_node);

    return vertex_no;
}

/* For the heap pointed to by h, this function decreases
 * the key of the node corresponding to vertex_no to new_value.
 * No check is made to ensure that new_value is in-fact
 * less than or equal to the current value,
 * so it is up to the user of this function
 * to ensure that this holds. */
void itth_decrease_key(ittheap_t *h,
                      int vertex_no, int new_value)
{
    ittheap_node_t *cut_node, *parent, *child, *next;
    int index, prevIndex;
    int inTree;
    int r;

    /* Obtain a pointer to the decreased node
     and its parent and child.*/
    cut_node = h->nodes[vertex_no];
    parent = cut_node->parent;
    index = cut_node->key % (C + 1);
    prevIndex = index;
    cut_node->key = new_value;

    /* Remove the decreased node from the current position
     in the S array. */
    if (cut_node->prev != NULL) {
        (cut_node->prev)->next = cut_node->next;
        inTree = 0;
    }
    else {
        (h->slist[index]).first_node = cut_node->next;
        inTree = 1;
    }
    if (cut_node->next != NULL) {
        (cut_node->next)->prev = cut_node->prev;
    }
    else {
        (h->slist[index]).last_node = cut_node->prev;
    }

    /* Insert the decreased node into a new position. */
    index = new_value % (C + 1);

    /* If the node has been decreased to a value
     that does not equal key values of any other key. */

```

```

if ((h->slist[index]).first_node == NULL) {
    (h->slist[index]).first_node = cut_node;
    (h->slist[index]).last_node = cut_node;
    cut_node->prev = NULL;
    cut_node->next = NULL;

    /* Now remove the node and its tree and reinsert it. */
    if (inTree) {
        if (parent) {
            iremove_node(h, cut_node);
            cut_node->right = cut_node->left = NULL;
            itth_meld(h, cut_node);
        }
    }
    else {
        itth_meld(h, cut_node);
    }
}

/* If another node already has that key value,
the decreased node needs to be made inactive
if it was active.
i.e. removed from the tree. */
else {
    ((h->slist[index]).last_node->next = cut_node;
    cut_node->prev = (h->slist[index]).last_node;
    cut_node->next = NULL;
    (h->slist[index]).last_node = cut_node;

    /* Make the node inactive by removing it form the tree,
and melding all its children back to the tree. */
    if (inTree) {
        if (!parent) {
            r = cut_node->dim;
            h->trees[r] = NULL;
            h->value -= (1 << r);
            h->n--;
        }
        else {
            iremove_node(h, cut_node);
        }
        child = cut_node->child;
        if(child) {
            next = child->right;
            next->left = child->right = NULL;
            itth_meld(h, next);
        }
        cut_node->parent = NULL;
        cut_node->child = NULL;
    }
}

```

```

        cut_node->left = NULL;
        cut_node->right = NULL;
        cut_node->dim = 0;
    }
}
if (h->slist[prevIndex].first_node != NULL && inTree) {
    itth_meld(h, (h->slist[prevIndex]).first_node);
}
}

/**/ Definitions of functions only visible within this file. ***/

/* melds the linked list of trees pointed to by *tree_list into
 * the heap pointed to by h.
 * This function uses the 'right' sibling pointer of nodes
 * to traverse the linked list from lower dimension nodes
 * to higher dimension nodes.
 * It expects the last nodes 'right' pointer to be NULL. */
void itth_meld(ittheap_t *h, ittheap_node_t *tree_list)
{
    ittheap_node_t *next, *add_tree;
    ittheap_node_t *carry_tree;
    int d;

#ifdef ITTHEAP_DUMP
printf("meld - ");  fflush(stdout);
#endif

    /* add_tree points to the current tree to be merged. */
    add_tree = tree_list;
    carry_tree = NULL;
    do {
        /* add_tree() gets merged into the heap,
         * and also carry_tree if one
         * exists from a previous merge. */

        /* Keep a pointer to the next tree and remove sibling
         * and parent links from the current tree.
         * The dimension of the next tree is always
         * one greater than the dimension of the previous tree,
         * so this merging is like an addition of
         * two ternary numbers.

         * Note that if add_tree is NULL
         * and the loop has not exited,
         * then there is only a carry_tree to be merged,
         * so treat it like add_tree. */
        if(add_tree) {
            next = add_tree->right;
            add_tree->right = add_tree->left = add_tree;

```

```

        add_tree->parent = NULL;
    }
    else {
        add_tree = carry_tree;
        carry_tree = NULL;
    }

#if ITTHEAP_DUMP
    printf("%d, ", add_tree->vertex_no);  fflush(stdout);
#endif

    /* First we merge add_tree with carry_tree,
     * if there is one. Note that carry_tree contains
     * only one node in its main trunk,
     * and add_tree has at most two,
     * so the result is at most one 3-node trunk,
     * which is treated as a 1-node main trunk
     * one dimension higher up. */
    if(carry_tree) {
        h->key_comps += imerge(&add_tree, &carry_tree);
    }

    /* After the merge, if add_tree is NULL,
     * then the resulting tree pointed to by carry_tree
     * carries to higher entry,
     * so we do not need to merge anything
     * into the existing main trunk.
     * If add_tree is not NULL
     * we add it to the existing main trunk. */
    if(add_tree) {
        d = add_tree->dim;
        if(h->trees[d]) {
            /* Nodes already in this main trunk position,
             * so merge. */
            h->key_comps += imerge(&h->trees[d], &add_tree);
            if(!h->trees[d]) h->value -= (1 << d);
            carry_tree = add_tree;
        }
        else {
            /* No nodes in this main trunk position,
             * so use add_tree. */
            h->trees[d] = add_tree;
            h->value += (1 << d);
        }
    }

    /* Obtain a pointer to the next tree to add. */
    add_tree = next;

    /* We continue if there is still a node in the list

```

```

        * to be merged, or a carry tree remains to be merged. */
    } while(add_tree || carry_tree);

#if ITTHEAP_DUMP
    printf("meld-exited, "); fflush(stdout);
#endif
}

/* merges the two trunks pointed to by *a and *b,
 * returning the sum trunk through 'a'
 * and any carry tree through 'b'.
 * When this function is used, both parameters 'a' and 'b'
 * refer to either a 1-node or 2-node trunk.
 * Returns the number of key comparisons used. */
int imerge(ittheap_node_t **a, ittheap_node_t **b)
{
    ittheap_node_t *tree, *next_tree, *other, *next_other;
    int c;

    /* Number of comparisons. */
    c = 0;

    /* 'tree' always points to the node with the lowest key.
     * To begin with, 'tree' points to the smaller head node,
     * and 'other' points to the head node of the other trunk. */
    if((*a)->key <= (*b)->key) {
        tree = (*a);
        other = (*b);
    }
    else {
        tree = (*b);
        other = (*a);
    }
    c++;

    /* next_tree points to the next node on the trunk
     * that 'tree' is the head of (if there is another node).
     * next_other points to the next node on the trunk
     * that 'other' is the head of (if there is another node). */
    next_tree = tree->child;
    if(next_tree && next_tree->dim != other->dim) {
        next_tree = NULL;
    }
    next_other = other->child;
    if(next_other && next_other->dim != other->dim) {
        next_other = NULL;
    }
}

```

```

/* The merging depends on the existence of nodes
   and the values of keys. */
if(!next_tree) {
    /* next_tree does not exist,
     * so we simply make 'other' the child of 'tree'.
     * If next_other exist
     * the resulting 3-node trunk is a carry tree. */

    iadd_child(tree, other);
    if(next_other) {
        tree->dim++;
        *a = NULL; *b = tree;
    }
    else {
        *a = tree; *b = NULL;
    }
}
else if(!next_other) {
    /* next_tree exists but next_other does not,
     * so the linked order of next_tree
     * and 'other' in the resulting 3-node trunk
     * depends on the values of keys.
     * The resulting 3-node trunk becomes a carry tree. */

    if(next_tree->key <= other->key) {
        iadd_child(next_tree, other);
    }
    else {
        ireplace_node(next_tree, other);
        iadd_child(other, next_tree);
    }
    c++;
    tree->dim++;
    *a = NULL; *b = tree;
}
else {
    /* Otherwise, both next_tree and next_other exist.
     * The result consists of a 1 node trunk
     * plus the 3-node trunk which becomes a carry tree.
     * We two trunks are made up as (tree, other, next_other)
     * and (next_tree). This uses no key comparisons. */

    ireplace_node(next_tree, other);
    next_tree->left = next_tree->right = next_tree;
    next_tree->parent = NULL;
    tree->dim++;
    *a = next_tree; *b = tree;
}
return c;

```

```

}

/* removes r_node, and the sub-tree it is the root of,
 * from the heap pointed to by h.
 * If necessary,
 * this causes rearrangement of r_node's work space. */
void iremove_node(ittheap_t *h, ittheap_node_t *r_node)
{
    ittheap_node_t *parent, *child;
    ittheap_node_t *ax, *bx, *ap, *bp, *b1, *c, *p;
    int d, d1;

    parent = r_node->parent;
    child = r_node->child;
    d = r_node->dim;

    /* If this node is an extra node
     * we simply cut the link between it and its
     * parent and update its sibling pointers. */
    if(d == parent->dim) {
        itrim_xnode(r_node);
    }

    /* Else if its child is an extra node
     then use its child to replace it. */
    else if(child && child->dim == d) {

        /* First we remove the child. */
        itrim_xnode(child);

        /* Now we put the child in r_node's position. */
        ireplace_node(r_node, child);
    }

    /* Otherwise we need some rearrangement of the workspace. */
    else {
        /* Look at up to two similar nodes in the work space
         * and determine if they have an extra node under them.
         * Nodes relative to the node being removed
         * are pointed to by the pointers ax, ap, bx, and bp.
         * If a similar trunk lies immediately below
         * cut_node's trunk in the work space,
         * then either ax or ap will be set to point to the node
         * on the end of that trunk.
         * The same applies for bx and bp,
         * but with a similar trunk immediately above
         * in the work space.
         * the 'x' pointers are set if there is a 3rd
         * node on the trunk.

```

```

* Otherwise the 'p' pointer is set to point
* to the 2nd node.
* Pointers will be set to null
* if a trunk does not exist or they are not used. */

/* Check for nodes on a similar trunk
above in the work space. */
p = r_node->parent->left;
if (p->dim == d) {
    c = p->child;
    if(c && c->dim == d) {
        ax = c;  ap = NULL;
    }
    else {
        ap = p;  ax = NULL;
    }
}
else {
    ax = ap = NULL;
}

/* Check for nodes on a similar trunk
below in the work space. */
d1 = d + 1;
p = r_node->right;
if (p->dim == d1) {
    p = p->child;
    if(p->dim == d1) p = p->left;

    c = p->child;
    if(c && c->dim == d) {
        bx = c;  bp = NULL;
    }
    else {
        bp = p;  bx = NULL;
    }
}
else {
    bx = bp = NULL;
}

if(bx) {

    /* First break 'bx's parent link
and sibling links. */
    itrim_xnode(bx);

    /* Then we insert bx in r_nodes place. */
    ireplace_node(r_node, bx);
}

```

```

else if(bp) {

    b1 = bp->parent;

    /* Recursively remove b1. */
    iremove_node(h, b1);
    b1->dim = d;

    ireplace_node(r_node, b1);

    /* It may improve speed by using trim_xnode()
       when recursion can be avoided. */
}
else if(ax) {

    /* Bend the tree to modify its shape
       then remove r_node. */
    iswap_trunks(ax->parent, parent);
    itrim_xnode(r_node);
}
else if(ap) {

    /* Bend the tree, so that the node to be relocated,
       parent, has the larger key value. */
    if(parent->key < ap->key) {
        iswap_trunks(ap, parent);
        p = parent;
        parent = ap;
        ap = p;
    }
    h->key_comps++;

    itrim_xnode(r_node);
    iremove_node(h, parent);
    parent->dim = d;

    /* Make parent the child of ap. */
    iadd_child(ap, parent);
}
else {
    /* The work space only has r_node node and parent.
       * This only occurs when parent is a root node,
       * so after removing r_node we demote parent
       * to a lower dimension main trunk. */

    /* Note that parent is a root node
       and has dimension d + 1. */
    h->trees[d+1] = NULL;
    h->value -= (1 << (d+1));
}

```

```

        parent->dim = d;
        itrim_xnode(r_node);
        parent->left = parent->right = NULL;

        itth_meld(h, parent);
    }
}

/* trims an extra node, x, from its trunk. */
void itrim_xnode(ittheap_node_t *x)
{
    ittheap_node_t *l, *r;

    if(x->dim == 0) {
        /* A dimension 0 node is an only child,
         * so cutting it leaves no children. */

        x->parent->child = NULL;
    }
    else {
        /* Otherwise, sibling pointers of other child nodes
         * must be updated. */
        l = x->left;
        r = x->right;
        l->right = r;
        r->left = l;
        x->parent->child = l;
    }
}

/* Where a node in an (i)th trunk, tr_low, and a node in an
 * (i+1)th trunk, tr_high, share the same parent,
 * this function is used for swapping them. */
void iswap_trunks(ittheap_node_t *tr_low,
                 ittheap_node_t *tr_high)
{
    int d;
    ittheap_node_t *parent, *l, *r;

    /* The dimensions of the two nodes are exchanged. */
    d = tr_low->dim;
    tr_low->dim = tr_high->dim;
    tr_high->dim = d;

    /* Obtain a pointer to the parent of both nodes. */
    parent = tr_high->parent;

    /* If the left sibling of tr_low is not tr_high,
     * we need to update sibling pointers.

```

```

    * Otherwise, the child pointer of the common parent now
    * points to tr_low. */
    if((l = tr_low->left) != tr_high) {

        /* Update sibling pointers. */
        r = tr_high->right;
        tr_high->left = l;
        tr_low->right = r;
        tr_high->right = tr_low;
        tr_low->left = tr_high;
        l->right = tr_high;
        r->left = tr_low;

        /* Determine if the child pointer of the common parent
        will need to be updated. */
        if(parent->child == tr_high) {
            parent->child = tr_low;
        }
    }
    else {
        parent->child = tr_low;
    }
}

/* makes node c and its tree a child of node p. */
void iadd_child(ittheap_node_t *p, ittheap_node_t *c)
{
    ittheap_node_t *l, *r;

    /* If p already has child nodes
    * we must update the sibling pointers.
    * Otherwise only initialise the left and right pointers
    * of the added child. */
    if((l = p->child)) {
        r = l->right;
        c->left = l;
        c->right = r;
        r->left = c;
        l->right = c;
    }
    else {
        c->left = c->right = c;
    }

    p->child = c;
    c->parent = p;
}

/* replaces node 'old' and its sub-tree with node 'new' and

```

```

    * its sub-tree. */
void ireplace_node(ittheap_node_t *old, ittheap_node_t *new)
{
    ittheap_node_t *parent, *l, *r;

    l = old->left;
    r = old->right;

    /* If 'old' is an only child
     * we only need to initialise the sibling pointers
     * of the new node.
     * Otherwise we update sibling pointers
     * of other child nodes. */
    if(r == old) {
        new->right = new->left = new;
    }
    else {
        l->right = new;
        r->left = new;
        new->left = l;
        new->right = r;
    }

    /* Update parent pointer of the new node
     * and possibly the child pointer of the parent node. */
    parent = old->parent;
    new->parent = parent;
    if(parent->child == old) parent->child = new;
}

/** Debugging Functions */

/* Recursively print the nodes of a 2-3 heap. */
#ifdef ITTHEAP_DUMP
void itth_dump_nodes(ittheap_node_t *ptr, int level)
{
    ittheap_node_t *child_ptr, *partner;
    int i, ch_count;

    /* Print leading whitespace for this level. */
    for(i = 0; i < level; i++) printf("  ");

    printf("%d(%ld)\n", ptr->vertex_no, ptr->key);

    if((child_ptr = ptr->child)) {
        child_ptr = ptr->child->right;

        ch_count = 0;

        do {

```

```

        itth_dump_nodes(child_ptr, level+1);
        if(child_ptr->dim != ch_count) {
            for(i = 0; i < level+1; i++) printf(" ");
            printf("error(dim)\n"); exit(1);
        }
        if(child_ptr->parent != ptr) {
            for(i = 0; i < level+1; i++) printf(" ");
            printf("error(parent)\n");
        }
        child_ptr = child_ptr->right;
        ch_count++;
    } while(child_ptr != ptr->child->right);

    if(ch_count != ptr->dim && ch_count != ptr->dim + 1) {
        for(i = 0; i < level; i++) printf(" ");
        printf("error(ch_count)\n"); exit(1);
    }
}
else {
    if(ptr->dim != 0) {
        for(i = 0; i < level; i++) printf(" ");
        printf("error(dim)\n"); exit(1);
    }
}
}
#endif

/* Print out a 2-3 heap. */
#if ITTHEAP_DUMP
void itth_dump(ittheap_t *h)
{
    int i;
    ittheap_node_t *ptr;

    printf("\n");
    printf("value = %d\n", h->value);
    printf("array entries 0..max_trees =");
    for(i=0; i<h->max_trees; i++) {
        printf(" %d", h->trees[i] ? 1 : 0 );
    }
    printf("\n\n");
    for(i=0; i<h->max_trees; i++) {
        if((ptr = h->trees[i])) {
            printf("tree %d\n\n", i);
            itth_dump_nodes(ptr, 0);
            printf("\n");
        }
    }
}
fflush(stdout);

```

```

}
#endif

/** Implement the universal heap structure type */

/* 2-3 heap wrapper functions. */

int _itth_delete_min(void *h) {
    return itth_delete_min((ittheap_t *)h);
}

void _itth_insert(void *h, int v, long k) {
    itth_insert((ittheap_t *)h, v, k);
}

void _itth_decrease_key(void *h, int v, long k) {
    itth_decrease_key((ittheap_t *)h, v, k);
}

int _itth_n(void *h) {
    return ((ittheap_t *)h)->n;
}

long _itth_key_comps(void *h) {
    return ((ittheap_t *)h)->key_comps;
}

void *_itth_alloc(int n) {
    return itth_alloc(n);
}

void _itth_free(void *h) {
    itth_free((ittheap_t *)h);
}

void _itth_dump(void *h) {
#ifdef ITTHEAP_DUMP
    itth_dump((ittheap_t *)h);
#endif
}

/* 2-3 heap info. */
const heap_info_t ITTHEAP_info = {
    _itth_delete_min,
    _itth_insert,
    _itth_decrease_key,
    _itth_n,
    _itth_key_comps,
    _itth_alloc,
    _itth_free,

```

```
    _itth_dump
};
```

## Linear Bucket System

```
#include <stdlib.h>
#if BUCKET_DUMP
#include <stdio.h>
#endif
#include "bucket.h"
#include "../dijkstra/edgeCost.h"

/* creates and returns a pointer to a linear bucket system.
   Argument max_nodes specifies the maximum number of nodes
   the system can contain. */
l_buckets_t *bucket_alloc(int max_nodes)
{
    int i;
    l_buckets_t *h;

    /* allocate memory for the linear bucket system. */
    h = malloc(sizeof(l_buckets_t));

    /* The maximum number of nodes. */
    h->max_nodes = max_nodes;

    /* Array of buckets for each value of edge cost,
       and array of pointers to nodes
       in the linear bucket system. */
    h->buckets = calloc(C + 1, sizeof(bucket_t));
    h->nodes = calloc(max_nodes, sizeof(bucket_node_t *));

    /* Begin with no nodes in the heap. */
    h->n = 0;

    /* Number of key_comparisons for experimental purposes. */
    h->key_comps = 0;

    /* The position where the node with
       minimum key value is stored in. */
    h->min_pos = 0;

    return h;
}

/* bucket_free() - destroys the buckets pointed to by h,
   freeing up any space that was used by it.
*/
void bucket_free(l_buckets_t *h)
{

```

```

        int i;

    #if BUCKET_DUMP
        printf("bucket_free started\n");
    #endif

        for (i = 0; i < h->max_nodes; i++) {
            free(h->nodes[i]);
        }

        free(h->nodes);
        free(h->buckets);
        free(h);

    #if BUCKET_DUMP
        printf("bucket_free finished\n");
    #endif

}

/* creates and inserts a new node representing vertex_no
   with key k into the heap pointed to by h. */
void bucket_insert(l_buckets_t *h, int vertex_no, int k)
{
    int index;
    bucket_node_t *new;

    #if BUCKET_DUMP
        printf("bucket_insert started\n");
    #endif

    /* Create and initialise the new node. */
    new = malloc(sizeof(bucket_node_t));
    new->vertex_no = vertex_no;
    new->key = k;
    new->prev = NULL;
    new->next = NULL;

    /* Maintain a pointer to vertex_no's
       new node in the buckets. */
    h->nodes[vertex_no] = new;

    /* Inserts the new node into the corresponding bucket. */
    index = k % (C + 1);

    if ((h->buckets[index]).first_node == NULL) {
        (h->buckets[index]).first_node = new;
        (h->buckets[index]).last_node = new;
    }
    else {

```

```

        ((h->buckets[index]).last_node)->next = new;
        new->prev = (h->buckets[index]).last_node;
        (h->buckets[index]).last_node = new;
    }

    /* Update the heap's node count. */
    h->n++;

#if BUCKET_DUMP
    printf("bucket_insert finished\n");
#endif

}

/* deletes the minimum node from the bucket
   pointed to by h and returns its vertex number. */
int bucket_delete_min(l_buckets_t *h)
{
    bucket_node_t *min_node;
    int vertex_no;

#if BUCKET_DUMP
    printf("bucket_delete_min started\n");
#endif

    /* Find the min_node by traversing the array of buckets. */
    min_node = (h->buckets[h->min_pos]).first_node;

    while (min_node == NULL) {
        h->min_pos++;
        h->min_pos = h->min_pos % (C + 1);
        min_node = (h->buckets[h->min_pos]).first_node;
    }

    /* After min_node has been found,
       remove it from the bucket. */
    (h->buckets[h->min_pos]).first_node =
        ((h->buckets[h->min_pos]).first_node)->next;

    if ((h->buckets[h->min_pos]).first_node == NULL) {
        (h->buckets[h->min_pos]).last_node = NULL;
    }
    else {
        ((h->buckets[h->min_pos]).first_node)->prev = NULL;
    }

    /* Record the vertex_no to return. */
    vertex_no = min_node->vertex_no;

    /* Delete the minimum node from the array of nodes. */

```

```

        h->nodes[vertex_no] = NULL;
        free(min_node);
        h->n--;

#ifdef BUCKET_DUMP
        printf("bucket_delete_min finished\n");
#endif

        return vertex_no;
    }

/* For the heap pointed to by h,
   this function decreases the key of the node
   corresponding to vertex_no to new_value.
   No check is made to ensure that new_value
   is in-fact less than or equal to the current value,
   so it is up to the user of this function
   to ensure that this holds. */
void bucket_decrease_key(l_buckets_t *h,
                        int vertex_no, int new_value)
{
    int index;
    bucket_node_t *decreased_node;

#ifdef BUCKET_DUMP
    printf("bucket_decrease_key on vn= %d\n", vertex_no);
    printf("from %d to %d\n", (h->nodes[vertex_no])->key,
           new_value);
#endif

    /* Obtain a pointer to the decreased node. */
    decreased_node = h->nodes[vertex_no];
    index = decreased_node->key % (C + 1);
    decreased_node->key = new_value;

    /* Re-organise the doubly linked list. */
    if (decreased_node->prev != NULL) {
        (decreased_node->prev)->next = decreased_node->next;
    }
    else {
        (h->buckets[index]).first_node = decreased_node->next;
    }
    if (decreased_node->next != NULL) {
        (decreased_node->next)->prev = decreased_node->prev;
    }
    else {
        (h->buckets[index]).last_node = decreased_node->prev;
    }

    /* Insert the decreased node into a new position. */

```

```

    index = new_value % (C + 1);

    if ((h->buckets[index]).first_node == NULL) {
        (h->buckets[index]).first_node = decreased_node;
        (h->buckets[index]).last_node = decreased_node;
        decreased_node->prev = NULL;
        decreased_node->next = NULL;
    }
    else {
        ((h->buckets[index]).last_node)->next = decreased_node;
        decreased_node->prev = (h->buckets[index]).last_node;
        decreased_node->next = NULL;
        (h->buckets[index]).last_node = decreased_node;
    }

#if BUCKET_DUMP
    printf("bucket_decrease_key finished\n");
#endif

}

/** Implement the universal heap structure type */

/* Linear bucket system wrapper functions. */

int _bucket_delete_min(void *h) {
    return bucket_delete_min((l_buckets_t *)h);
}

void _bucket_insert(void *h, int v, long k) {
    bucket_insert((l_buckets_t *)h, v, k);
}

void _bucket_decrease_key(void *h, int v, long k) {
    bucket_decrease_key((l_buckets_t *)h, v, k);
}

int _bucket_n(void *h) {
    return ((l_buckets_t *)h)->n;
}

long _bucket_key_comps(void *h) {
    return ((l_buckets_t *)h)->key_comps;
}

void *_bucket_alloc(int n) {
    return bucket_alloc(n);
}

void _bucket_free(void *h) {

```

```

        bucket_free((l_buckets_t *)h);
    }

void _bucket_dump(void *h) {
    #if BUCKET_DUMP
        bucket_dump((l_buckets_t *)h);
    #endif
}

/* 2-3 heap info. */
const heap_info_t BUCKET_info = {
    _bucket_delete_min,
    _bucket_insert,
    _bucket_decrease_key,
    _bucket_n,
    _bucket_key_comps,
    _bucket_alloc,
    _bucket_free,
    _bucket_dump
};

```

## One Level Index

```

#include <stdlib.h>
#include <math.h>
#if MBUCKET_DUMP
#include <stdio.h>
#endif
#include "mbucket.h"
#include "../dijkstra/edgeCost.h"

/* creates and returns a pointer to the bucket system.
   Argument max_nodes specifies the maximum number of nodes
   the system can contain. */
ml_buckets_t *mbucket_alloc(int max_nodes)
{
    int i;
    int temp;

    ml_buckets_t *h;

    /* allocate memory for the linear bucket system. */
    h = malloc(sizeof(ml_buckets_t));

    /* The maximum number of nodes. */
    h->max_nodes = max_nodes;

    /* sqrtC = sqrt(C) */
    h->sqrtC = ((int) sqrt((double) C)) + 1;
}

```

```

    /* Array that keeps count of the number of nodes
       in a given range of buckets. */
    h->index = calloc(h->sqrtC, sizeof(int));
    for (i = 0; i < h->sqrtC; i++) {
        h->index[i] = 0;
    }

    /* Array of buckets for each value of edge cost,
       and array of pointers to nodes
       in the linear bucket system. */
    h->buckets = calloc(C + 1, sizeof(mbucket_t));
    h->nodes = calloc(max_nodes, sizeof(mbucket_node_t *));

    /* Begin with no nodes in the heap. */
    h->n = 0;

    /* Number of key_comparisons for experimental purposes. */
    h->key_comps = 0;

    /* The position where the node with
       minimum key value is stored in. */
    h->min_pos = 0;

    return h;
}

/* bucket_free() - destroys the buckets pointed to by h,
   freeing up any space that was used by it.
*/
void mbucket_free(ml_buckets_t *h)
{
    int i;

#ifdef MBUCKET_DUMP
    printf("bucket_free started\n");
#endif

    for (i = 0; i < h->max_nodes; i++) {
        free(h->nodes[i]);
    }
    free(h->nodes);
    free(h->buckets);
    free(h->index);
    free(h);

#ifdef MBUCKET_DUMP
    printf("bucket_free finished\n");
#endif
}

```

```

/* creates and inserts a new node representing vertex_no
   with key k into the heap pointed to by h. */
void mbucket_insert(ml_buckets_t *h, int vertex_no, int k)
{
    int i;
    int index;
    int logIndex;
    mbucket_node_t *new;

#ifdef BUCKET_DUMP
    printf("bucket_insert started\n");
#endif

    /* Create and initialise the new node. */
    new = malloc(sizeof(mbucket_node_t));
    new->vertex_no = vertex_no;
    new->key = k;
    new->prev = NULL;
    new->next = NULL;

    /* Maintain a pointer to vertex_no's
       new node in the buckets. */
    h->nodes[vertex_no] = new;

    /* Inserts the new node into the corresponding bucket. */
    index = k % (C + 1);

    if ((h->buckets[index]).first_node == NULL) {
        (h->buckets[index]).first_node = new;
        (h->buckets[index]).last_node = new;
    }
    else {
        ((h->buckets[index]).last_node)->next = new;
        new->prev = (h->buckets[index]).last_node;
        (h->buckets[index]).last_node = new;
    }

    /* Update the heap's node count. */
    h->n++;

    /* increment the corresponding index by 1. */
    h->index[index / h->sqrtC]++;

#ifdef BUCKET_DUMP
    printf("bucket_insert finished\n");
#endif
}

```

```

/* deletes the minimum node from the bucket
   pointed to by h and returns its vertex number. */
int mbucket_delete_min(ml_buckets_t *h)
{
    int vertex_no;
    int nextIndex;
    int nextSection;
    mbucket_node_t *min_node;

#ifdef BUCKET_DUMP
    printf("bucket_delete_min started\n");
#endif

    nextSection = (h->min_pos / h->sqrtC) + 1;
    nextIndex = nextSection * h->sqrtC;

    /* Find the min_node by traversing the array of buckets. */
    min_node = (h->buckets[h->min_pos]).first_node;

    while (min_node == NULL) {
        h->min_pos++;
        if (h->min_pos == nextIndex) {
            while (h->index[nextSection] == 0) {
                nextSection = nextSection + 1;
                nextSection = nextSection % h->sqrtC;
            }
            h->min_pos = nextSection * h->sqrtC;
        }
        h->min_pos = h->min_pos % (C + 1);
        min_node = (h->buckets[h->min_pos]).first_node;
    }

    /* After min_node has been found,
       remove it from the bucket. */
    (h->buckets[h->min_pos]).first_node =
        ((h->buckets[h->min_pos]).first_node)->next;

    if ((h->buckets[h->min_pos]).first_node == NULL) {
        (h->buckets[h->min_pos]).last_node = NULL;
    }
    else {
        ((h->buckets[h->min_pos]).first_node)->prev = NULL;
    }

    h->index[h->min_pos / h->sqrtC]--;

    /* Record the vertex_no to return. */
    vertex_no = min_node->vertex_no;

```

```

        /* Delete the minimum node from the array of nodes. */
        h->nodes[vertex_no] = NULL;
        free(min_node);
        h->n--;

#if BUCKET_DUMP
        printf("bucket_delete_min finished\n");
#endif

        return vertex_no;
}

/* For the heap pointed to by h,
   this function decreases the key of the node
   corresponding to vertex_no to new_value.
   No check is made to ensure that new_value
   is in-fact less than or equal to the current value,
   so it is up to the user of this function
   to ensure that this holds. */
void mbucket_decrease_key(ml_buckets_t *h,
                          int vertex_no, int new_value)
{
    int index;
    mbucket_node_t *decreased_node;

#if BUCKET_DUMP
    printf("bucket_decrease_key on vn= %d\n", vertex_no);
    printf("from %d to %d\n",
           (h->nodes[vertex_no])->key, new_value);
#endif

    /* Obtain a pointer to the decreased node. */
    decreased_node = h->nodes[vertex_no];
    index = decreased_node->key % (C + 1);
    decreased_node->key = new_value;

    h->index[index / h->sqrtC]--;

    /* Re-organise the doubly linked list. */
    if (decreased_node->prev != NULL) {
        (decreased_node->prev)->next = decreased_node->next;
    }
    else {
        (h->buckets[index]).first_node = decreased_node->next;
    }
    if (decreased_node->next != NULL) {
        (decreased_node->next)->prev = decreased_node->prev;
    }
    else {

```

```

        (h->buckets[index]).last_node = decreased_node->prev;
    }

    /* Insert the decreased node into a new position. */
    index = new_value % (C + 1);

    if ((h->buckets[index]).first_node == NULL) {
        (h->buckets[index]).first_node = decreased_node;
        (h->buckets[index]).last_node = decreased_node;
        decreased_node->prev = NULL;
        decreased_node->next = NULL;
    }
    else {
        ((h->buckets[index]).last_node->next = decreased_node;
        decreased_node->prev = (h->buckets[index]).last_node;
        decreased_node->next = NULL;
        (h->buckets[index]).last_node = decreased_node;
    }

    h->index[index / h->sqrtC]++;

#ifdef BUCKET_DUMP
    printf("bucket_decrease_key finished\n");
#endif

}

/** Implement the universal heap structure type */

/* Linear bucket system wrapper functions. */

int _mbucket_delete_min(void *h) {
    return mbucket_delete_min((ml_buckets_t *)h);
}

void _mbucket_insert(void *h, int v, long k) {
    mbucket_insert((ml_buckets_t *)h, v, k);
}

void _mbucket_decrease_key(void *h, int v, long k) {
    mbucket_decrease_key((ml_buckets_t *)h, v, k);
}

int _mbucket_n(void *h) {
    return ((ml_buckets_t *)h)->n;
}

long _mbucket_key_comps(void *h) {
    return ((ml_buckets_t *)h)->key_comps;
}

```

```

void *_mbucket_alloc(int n) {
    return mbucket_alloc(n);
}

void _mbucket_free(void *h) {
    mbucket_free((ml_buckets_t *)h);
}

void _mbucket_dump(void *h) {
#ifdef BUCKET_DUMP
    mbucket_dump((ml_buckets_t *)h);
#endif
}

/* 2-3 heap info. */
const heap_info_t MBUCKET_info = {
    _mbucket_delete_min,
    _mbucket_insert,
    _mbucket_decrease_key,
    _mbucket_n,
    _mbucket_key_comps,
    _mbucket_alloc,
    _mbucket_free,
    _mbucket_dump
};

```

## Dijkstra's Algorithm

```

#include <stdlib.h>
#include "da.h"
#include "../timing/timing.h"
#include "../graphs/dgraph.h"

/** Special values. */
#define TRUE 1
#define FALSE 0

/* heap_dijkstra() - Heap implementation of Dijkstra's algorithm.
 * Requires a pointer, g, to the directed graph used,
 * a pointer to the starting vertex,
 * and a pointer to a da_heap_info_t structure for the heap used.
 * Returns a da_result_t structure
 * containing the resulting shortest path distances,
 * and timing information.
 */
da_result_t *heap_dijkstra(const dgraph_t *g, int v0,
                          const heap_info_t *heap_info)
{
    int v, w;

```

```

int *f, *s;
long dist, *d;
int n;

dgraph_vertex_t *vertices;
dgraph_edge_t *edge_ptr;

void *front;
int (*heap_delete_min)(void *);
void (*heap_insert)(void *, int, long);
void (*heap_decrease_key)(void *, int, long);
int (*heap_n)(void *);
void (*heap_alloc)(int);
void (*heap_free)(void *);

da_result_t *result;
long dist_comps;

size_t int_size, long_size;

/* Initialisation not specifically part of
   Dijkstra's algorithm. */
int_size = sizeof(int);
long_size = sizeof(long);
heap_delete_min = heap_info->delete_min;
heap_insert = heap_info->insert;
heap_decrease_key = heap_info->decrease_key;
heap_n = heap_info->n;
heap_alloc = heap_info->alloc;
heap_free = heap_info->free;

/* Start of Dijkstra's algorithm. */
dist_comps = 0;
timer_start();
vertices = g->vertices;
result = malloc(sizeof(da_result_t));
n = result->n = g->n;
d = result->d = calloc(n, long_size);
f = calloc(n, int_size);
s = calloc(n, int_size);
front = heap_alloc(n);

/* The start vertex is part of the solution set. */
s[v0] = TRUE;
d[v0] = 0;

/* Put out set of the starting vertex
   into the frontier and update the distances
   to vertices in the out set.
   k is the index for the out set. */

```

```

    edge_ptr = vertices[v0].first_edge;
    while(edge_ptr) {
        w = edge_ptr->vertex_no;
        dist = d[w] = edge_ptr->dist;
        heap_insert(front, w, dist);
#if DA_HEAP_DUMP
        heap_info->dump(front);
#endif
        f[w] = TRUE;
        edge_ptr = edge_ptr->next;
    }

    /* At this point we are assuming that
       all vertices are reachable from the
       starting vertex and N > 1 so that j > 0. */

    while(heap_n(front) > 0) {
        /* Find the vertex in frontier
           that has minimum distance. */
        v = heap_delete_min(front);

#if DA_HEAP_DUMP
        heap_info->dump(front);
#endif

        /* Move this vertex from the frontier
           to the solution set. */
        s[v] = TRUE;
        f[v] = FALSE;

        /* Update distances to vertices, w,
           in the out set of v. */
        edge_ptr = vertices[v].first_edge;
        while(edge_ptr) {
            w = edge_ptr->vertex_no;

            /* Only update if w is not already
               in the solution set. */
            if(!s[w]) {
                /* If w is in the frontier
                   the new distance to w is the minimum
                   of its current distance
                   and the distance to w via v. */
                dist = d[v] + edge_ptr->dist;
                if(f[w]) {
                    /* dist_comps++; */
                    if(dist < d[w]) {
                        d[w] = dist;
                        heap_decrease_key(front, w, dist);
                    }
                }
            }
        }
    }
#endif

```

```

                                heap_info->dump(front);
#endif
                                }
                                }
                                else {
                                    d[w] = dist;
                                    heap_insert(front, w, dist);
#if DA_HEAP_DUMP
                                    heap_info->dump(front);
#endif
                                f[w] = TRUE;
                                }
                                } /* if */
                                edge_ptr = edge_ptr->next;
                                } /* while */
                                } /* while */

/* End of Dijkstra's algorithm. */

/* Record timing information. */
result->ticks = timer_stop();
result->key_comps = dist_comps + heap_info->key_comps(front);

/* Free space used by arrays local to this function. */
free(f);
free(s);
heap_free(front);

return result;
}

/* frees up space used by a da_result_t structure. */
void da_result_free(da_result_t *r)
{
    free(r->d);
    free(r);
}

```

## Experiment Program

```

#include <stdio.h>
#include <stdlib.h>
#include "da.h"
#include "../graphs/dgraph.h"

/* #include all heaps to be tested using dijkstra's algorithm. */
#include "../heaps/bheap.h"
#include "../heaps/ttheap.h"
#include "../heaps/bucket.h"
#include "../heaps/ittheap.h"

```

```

#include "../heaps/mbucket.h"

/* This program generates key comparisons data
 * and/or CPU time data,
 * by selecting one of the following with a 1. */
#define KEY_COMPS_DATA 0
#define CPU_TIME_DATA 1

/* Step size between values of n used. */
#define STEP 10000

/* Structure type for summing the results for each heap. */
typedef struct timestruct {
    /* Heap description (i.e. name) */
    char *desc;
    /* Heap functions (passed to algorithm). */
    const heap_info_t *fns;
    /* For summing results of algorithm. */
    da_result_t sum;
} timestruct_t;

/* An array of time info structures holds the information
for each dictionary. */
timestruct_t heap_times[] = {
    { "2-3", &TTHEAP_info },
    { "Bucket", &BUCKET_info },
    { "Integer 2-3", &ITHEAP_info },
    { "2D Bucket", &MBUCKET_info }
};

int main(int argc, char *argv[])
{
    int i, j, k, n_max, n_samples, n_heaps;
    double edge_f;
    da_result_t *r;
    dgraph_t *graph;

    /* Process command line arguments (if any),
 * otherwise get input from the user.
 * Arguments supplied correspond to:
 * - number of samples used for average calculation.
 * - maximum graph size to test.
 * - edge factor (the average size of the out set).
 */
    if(argc == 4) {
        n_samples = atoi(argv[1]);
        n_max = atoi(argv[2]);
        edge_f = atof(argv[3]);
    }
    else {

```

```

        /* Several samples may be needed
           to calculate the time to the required accuracy. */
        printf("Enter the number samples to use: ");
        scanf("%d", &n_samples);

        /* We have an upper limit on the graph size tested. */
        printf("Enter the maximum value of n to use: ");
        scanf("%d", &n_max);

        /* The edge factor entered should be
           greater than 1 - 1/n */
        printf("Enter the edge factor
           (i.e. average size of OUT set): ");
        scanf("%lf", &edge_f);
        putchar('\n');
    }

    /* Number of heaps being tested. */
    n_heaps = sizeof(heap_times)/sizeof(timestruct_t);

    #if KEY_COMPS_DATA
        printf("Number of Comparisons for Dijkstra's Algorithm.\n");
    #endif

    #if CPU_TIME_DATA
        printf("CPU Time for Dijkstra's Algorithm (msec)\n");
    #endif

    printf("Graph Size = n, Edge factor = %.3f,
           Number of samples = %d\n", edge_f, n_samples);

    /* Print column labels */
    printf("\nResults:\n\n");
    for(j = 0; j < n_heaps; j++) {
        printf(",\t%s", heap_times[j].desc);
    }
    putchar('\n');

    /* Test over varying graph sizes. */
    for(k = STEP; k <= n_max; k += STEP) {
        /* Initialize the sums used
           in average calculations to zero. */
        for(j = 0; j < n_heaps; j++) {
            heap_times[j].sum.key_comps = 0;
            heap_times[j].sum.ticks = 0;
        }
        /* For the average calculation,
           sum the number of comparisons. */
        for(i = 0; i < n_samples; i++) {

```

```

    /* We use a new random graph for each run
       of Dijkstra's algorithm. */
    graph = dgraph_rnd_sparse(k, edge_f);

    /* Time each heap.
       * The heap_info structure of each heap
       * is passed to dijkstra's algorithm
       * in order to use the functions
       * provided by that heap. */
    for(j = 0; j < n_heaps; j++) {
        r = heap_dijkstra(graph, StartVertex,
                          heap_times[j].fns);
        heap_times[j].sum.key_comps += r->key_comps;
        heap_times[j].sum.ticks += r->ticks;
        da_result_free(r);
    }
    /* Free the random graph. */
    dgraph_free(graph);
}

#if KEY_COMPS_DATA
    /* Print line of key comparison results
       for current value of k. */
    printf("%d", k);
    for(j = 0; j < n_heaps; j++)
        printf("\t%.2f",
              (double)heap_times[j].sum.key_comps /
              n_samples);
    putchar('\n');
#endif

#if CPU_TIME_DATA
    /* Print line of key comparison results
       for current value of k. */
    printf("%d", k);
    for(j = 0; j < n_heaps; j++)
        printf("\t%.2f",
              (((double)heap_times[j].sum.ticks /
              n_samples) / CLOCK_DIV) * 1000);
    putchar('\n');
#endif
}
return 0;
}

```