A New Algorithm and Data Structures for the All Pairs Shortest Path Problem

Mashitoh Binti Hashim

Department of Computer Science and Software Engineering

University of Canterbury

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy (PhD) in Computer Science

2013

Abstract

In 1985, Moffat-Takaoka (MT) algorithm was developed to solve the all pairs shortest path (APSP) problem. This algorithm manages to get time complexity of $O(n^2 \log n)$ expected time when the end-point independent model of probabilistic assumption is used. However, the use of a critical point introduced in this algorithm has made the implementation of this algorithm quite complicated and the running time of this algorithm is difficult to analyze. Therefore, this study introduces a new deterministic algorithm for the APSP that provides an alternative to the existing MT algorithm. The major advantages of this approach compared to the MT algorithm are its simplicity, intuitive appeal and ease of analysis. Moreover, the algorithm was shown to be efficient as the expected running time is the same $O(n^2 \log n)$. Performance of a good algorithm depends on the data structure used to speed up the operations needed by the algorithm such as insert, delete-min and decrease-key operations. In this study, two new data structures have been implemented, namely quaternary and dimensional heaps. In the experiment carried out, the quaternary heap that employed similar concept with the trinomial heap with a special insertion cache function performed better than the trinomial heap when the number of n vertices was small. Likewise, the dimensional heap data structure executed the decrease-key operation efficiently by maintaining the thinnest structure possible through the use of thin and thick edges, far surpassing the existing binary, Fibonacci and 2-3 heaps data structures when a special acyclic graph was used. Taken together all these promising findings, a new improved algorithm running on a good data structure can be implemented to enhance the computing accuracy and speed of todays computing machines

I lovingly dedicate this thesis to my family, especially...

to my husband, Mohd Izhair, for his love, his patience and supporting me each step of the way;

to my chidren, Rabbiatul Adawiyah and Muadz Aliman, for being that little sunshine in my life;

to my mother, Sofiah, for her love and encouragement;

and

to my late father, Hashim, who has been my constant source of inspiration.

Acknowledgements

I would like to acknowledge many people for helping me through the completion of this thesis. Firstly, I acknowledge with my sincere gratitude and appreciation the professional supervision from Prof. Dr. Tadao Takaoka. His attention to every detail and academic precision provided me the necessary direction and focus for my study. You are a great teacher who is not only the selfless giver but also the mentor of my life.

I also would like to thank Associate Professor Dr Mukundan for being my second supervisor. You are a great friend to have.

I thank Professor Peter Eades, and Professor Hirata for being my examiners, who provided encouraging and constructive feedback. It is no easy task, reviewing a thesis, and I am grateful for their thoughtful and detailed comments.

My heartfelt thanks are extended to my friends at UC, Amali, Saima, Dr. Ray Hidayat, Sagaya, Cihui (Huazhong University of Science and Technology, during his exchange program at UC), Jason, Ravi, Muhammad, Kapila, Boli, Ibrahim and many more. To Amali, thanks a lot. Without your support and advice, it would be hard for me to proceed. To Saima, we have shared many memories together. I really love your cooking!

I wish to thank for all technical support I received at UC, especially to Joff and Phil.

Thanks to my friends at Universiti Pendidikan Sultan Idris for all the help I have received, especially to Dr. Ramlah, Haslina, Dr. Norasikin, Dr. Che Soh, Rasyidi, Dr. Khairulanuar and Tuan Haji Zahar.

I am also grateful to many persons who shared their memories, experiences and always be there for my family, especially to my neighbors, Pam and Peter, Dr. Khairuddin and kak Ita, Kak Jah and abg Hassan, Dr. Fahmi and Niza, Dr. Norshidrah and abg Jaafar, Dr. Mazwin and abg Wan, Marina and abg Mizi, Mazlina and abg Dino, Sabariah and Haneef, Faizi and Rita, Dr. Suriani and abg. Don, Kak Julie and Dr. Husaini, and many more.

Thanks also for invaluable support I received from my mother, Sofiah and my siblings, Manisah, Mohd Fadzil, Marina, Mohd Azhar and Maziah and their families. Thanks for believing in me and remembering me in your constant prayers. To my mother in-law, Hatanah and my in-laws, Wan Zaiton, Rosman, Zuhailah, Hazrina, Hazir and Din, thank you as well.

To my wonderful children, thank you for bearing with me and being my greatest supporters. To my husband, my friend, mentor and critic, thank you for not letting me give up and giving me all the encouragement I needed to continue. I love you so much.

Lastly I would like to thank the Ministry of Higher Education in Malaysia for funded my scholarship. Also to Universiti Pendidikan Sultan Idris (UPSI), Malaysia for allowing me to pursue my studies especially to Cik Zarina who has made difficult things easy.

Last but not least, I would like to thank Harish Bhanderi for making a standard Cambridge University PhD/MPhil thesis latex template available to use.

My experience at the University of Canterbury has been an excellent one.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

a word-RAM model	a computation model that its memory consists of an unbounded sequence of registers which each holds an integer value or a real number
acyclic graph	a graph in which there are no cycles
amortized cost	the idea is that some expensive operations can increase the potential, so that later operations can be done cheaply due to the increased of potential
APSP	abbreviation of all pairs shortest path; the APSP problem is a problem to determine the shortest path from each vertex to every other vertices in a graph
asymptotic analysis	an analysis that examines how the efficiency of a program changes as the program's input size approaches infinity. It observes the growth rate of the program when different input sizes are given
average-case analysis	the running time is the expected time for an algorithm to solve the problem over all inputs of size n
branch	a path connecting two nodes
cycle	a special path is obtained from a vertex v back to itself in a graph
dense graph	when the number of edges, m, is close to n^2, where n is the number of vertices
directed graph	a graph that has all directed edges; each edge can be traversed only in a specific direction
edge	a pair of vertices
graph	a graph, G, is defined as a data structure that consists of a set of vertices or nodes, V, and a set of edges or arcs, E, $G = (V, E)$
main trunk	the highest level trunk that connects the highest dimension nodes
node	represent a vertex with its key value
path	a path is defined as a sequence of vertices in which each pair of successive vertices is connected by an edge. The first vertex in the path is called a start vertex or an origin vertex and the last vertex is known

	as the end vertex or a destination vertex
root	a node that is located at the top level of a tree and it has no parent node
sparse graph	a graph that has only a few edges such as $m = 2n$
SPSP	abbreviation of single pair shortest path; the SPSP problem is a problem to determine the shortest path from a source vertex to destination vertex in a graph
SSSP	abbreviation of single source shortest path; the SSSP problem is a problem to determine the shortest path from a source vertex to all vertices in a graph
the comparison-addition model	this model assumes that all input distance data consist of m real numbers
trunk	a path that connect several nodes; maximum number of nodes on a trunk applies for a different heap
undirected graph	the edges of the graph are drawn with no arrow; they can be traversed in either direction
unweighted graph	opposite of a weighted graph; it shows the existence of a connection between two nodes
weighted graph	a graph that has weighted edges
worst-case analysis	the total running time is the maximum time taken for an algorithm to solve the problem on any input of size n

1

Introduction

"Which way to go, Sam?", John asked his friend when they were going out for fishing. Sam took out his TomTom Global Positioning System (GPS), and selected Twizel as a destination from Christchurch. They were planning to go fishing at Mount Cook. "Dont worry. Just trust GPS. I am confident that this is the shortest way and we could reach there in 3 hours and 53 minutes,' replied Sam as he pointed to the estimated time to arrive displayed on the GPS screen. "Are you sure this is the shortest route?" asked John again. "Yes, I have selected the shortest route in distance, but not in time, as we can use off-road highways to reach there. Hope, we can save fuel and as well as enjoying sightseeing," replied Sam.

Figure 1.1: Example of route suggested by TomTom GPS

The above conversation is about using a system to help in finding the shortest route to go from one town to another. Two choices are given here; whether to arrive at the destination at the shortest possible time or to choose the shortest route, regardless of how long it takes to reach the desired destination. If the road network from Christchurch to Twizel were represented by a weighted graph, finding the above route would be a well-known application in a graph theory known as the shortest path problem.

The problem of finding the shortest route from Christchurch and Twizel above is specifically called the single pair shortest path problem (SPSP). Other variations of this problem are the single source shortest path problem (SSSP) and the all pairs shortest path problem (APSP). The SSSP is a problem to find the shortest paths from one location, called a source node, to all other nodes in the graph. If the shortest paths between every pair of nodes are sought, the problem is called the APSP.

Initially, the shortest path problem is a problem of finding the shortest path from a specific node s, called a source, to a second specified node, called a destination. The shortest path obtained might consist of a collection of edges comprising the shortest path from s. Figure 1.2 shows an example of a typical graph. The distance values or edge costs that connect one node to another are enclosed in parenthesis; for example, the distance value from node 0 to node 1 is 3.

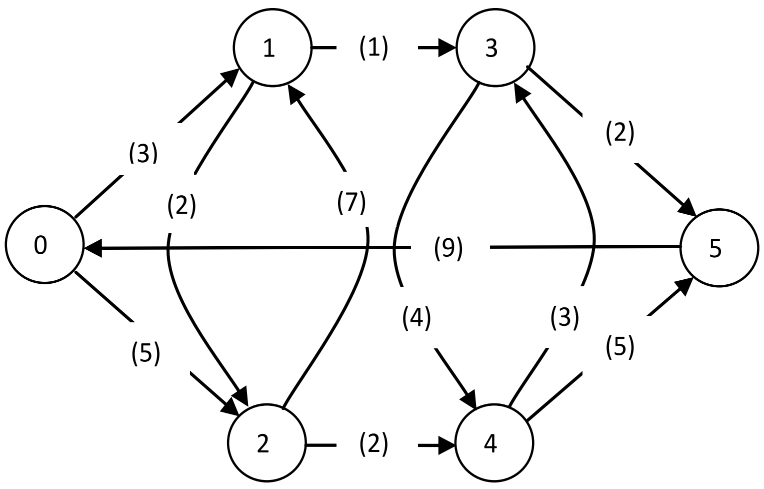


Figure 1.2: A typical network where $n = 6$ and $m = 11$

When all edge costs are non-negative, as shown in Figure 1.2, an efficient algorithm can be used to solve the problem. The shortest path algorithm is a program, or set of instructions that can be executed to provide the shortest path between locations. There are many algorithms which can be used to find the shortest distance. To solve the problem, the basic algorithm works like this. Starting with the source node s, the algorithm finds the shortest path from the source to the nearest node. The same steps are taken in subsequent iteration. The procedure requires a maximum of $m-1$ iterations to find the shortest path from s to another node where m is the total number of edges in the graph.

The edge cost can be any non-negative numbers. Recall the scenario discussed earlier. If we consider the time taken to arrive at the destination to be the most importan factor, then the edge cost represents the fastest time to go from one location to another. However, if the shortest route is required, the edge cost should be the shortest distance. In many other applications, the edge cost may represent capacity, length, speed, noise and so on.

Let us assume that Sam has two GPS systems which are different models and use two different algorithms. To compute the shortest distance from Christchurch to Twizel, the first GPS gives the result in two seconds, while the other takes five seconds. It is clear that the time taken varies even though the suggested routes are the same. In a case where there is no urgency, the time taken may not be considered. However, in an urgent case, for example if an ambulance crew needs tp fomd a route to pick up a critically ill patient, the second GPS might not be the right choice.

The exact speed of an algorithm depends on many factors such as implementation details, where the algorithm is run, and the input size to the algorithm. It is possible to use two algorithms to solve the same problem. The first algorithm has been executed on a powerful machine and the second one has been run on a slower machine. For the same amount of inputs, it was found that the first algorithm run on the powerful machine performs better compared to the second one. If both algorithms are executed on the same machine, using a small input size, the first algorithm solves the problem much faster, but in the case of a larger input size, the second algorithm is significantly superior. Table 1.1 illustrates different running times with different input sizes for these two algorithms. Both algorithms show that the running times comensurates with increase in input sizes. If the input size is relatively small, algorithm 1 is extremely fast. On the other hand, Algorithm 2 performs better when the input size is bigger. Now, which algorithm should we chose?

In answering this question, a proper algorithm analysis should be constructed. The purpose of algorithm analysis is not about determining how long does one algorithm take to solve one particular problem, but to see the performance of the algorithm when solving the same problem with different input sizes. In other words, it is about to see the growth rate of the running

Input Size	Algorithm 1	Algorithm 2
100	10^{-6} seconds	10^{-4} seconds
1000	10^{-3} seconds	10^{-1} seconds
10000	3 seconds	1 second
100000	3 minutes	10 seconds

Table 1.1: Different running times with different input sizes

time when the problem size increases. If the running time is the only issue (not the growth rate or the performance of the algorithm), then moving the algorithm to a faster computer makes the algorithm better, as it can complete the given task faster. If only a slower machine is available, this algorithm might not perform as well as expected. Therefore, algorithm analysis should be machine independent and performed without regard to any specific computer. In computer science, this analysis is known as asymptotic analysis. Asymptotic analysis evaluates the performance of an algorithm based on the given input size and not the actual run time on different machines.

As the input size or input data are both one of many factors that affect the running time of an algorithm, quick access the data is extremely important. The use of a good data structure here is equally important. A data structure is a particular way to store, search and retrieve data. It consists of a set of procedures for certain tasks. The data is organized and stored properly by the data structure for easy maintainance and accessibility.

The relationship between algorithm and data structure is just like that between employer and semployee. As an 'employer,' an algorithm does not have to worry about how the task is going to be carried out by the 'employee,' that is, the data structure. When an instruction is given to the data structure to do a certain task, the algorithm expects that the task will be done perfectly and efficiently as has been ordered. If the data structure used gives easier and faster access to the input data, the time taken to solve the problem will be significantly reduced. In a case when a large input size is involved and the use of memory is limited, the right data structure to choose is essential. As the data structure used by an algorithm can greatly affect the algorithm's performance, the design and implementation of good data structure are vital. Figure 1.3 shows the shortest path concept in general.

Heap data structure is used to help in solving the shortest path problem. A heap is a specialized tree based data structure where all elements are arranged accordingly to certain priority. The smallest or largest element normally will be placed on the top of the tree which is called a root level. Every single heap has its heap property to maintain the tree structure. A

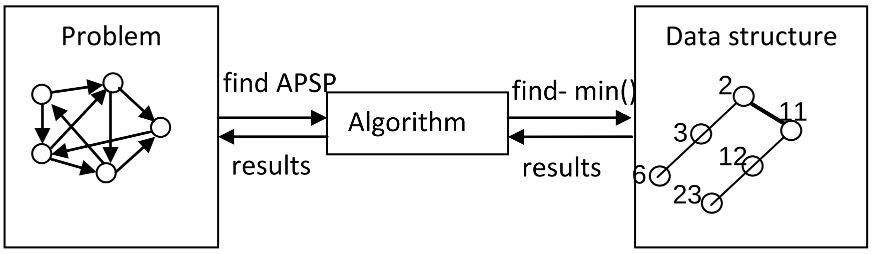


Figure 1.3: Relationship between shortest path problem, algorithm and data structure. Find-min is an example of operation used by the algorithm

heap property must be satisfied whenever changes are made to the elements in the heap, such as when a new element is inserted into the heap, the existing key element is changed, or when an element is removed from the heap. Chapter Three will discuss details about the tree or heap data structure.

Underlying the introductory sections are three objectives. Firstly, to introduce the shortest path problem; secondly, to reinforce the important of research in finding an efficient shortest path algorithm; and thirdly, to stress the importance of data structures to be used with the algorithm. The aim of this research is to develop a faster algorithm to solve the all pair shortest path problem and to develop good data structures that can be used to facilitate the process of finding the shortest path. The motivation of this thesis is to design the best shortest path algorithm to help find the shortest distance from one location to other locations. Even though this research has been carried out for many years, there are still many applications requiring an algorithm to solve related problem, especially during critical time such as when disaster strikes. Below are some scenarios that reveal the importance of this area of research.

A large explosion occurred in the Pike River mine in New Zealand in 2010. 31 miners were trapped and only two miners managed to escape. It was extremely dangerous to go into the mine looking for any survivors; instead, a robot was used. The robot needed to travel through certain points with a specific amount of battery charge. Since the robot was battery powered, and the distance to travel was uncertain, the robot should have been set up to go to the needed points in the most effective battery charge. The path followed needed to be the shortest one. The robot should have been capable of sending the fastest signal to the operator in order to update any finding. Again, a very good shortest path algorithm is needed to assist the operation. The robot controller also should be prepared to send two or three robots at the same time. Another shortest path algorithm was required. Nobody knew whether a clever shortest path algorithm was used to program the robot. In fact, the rescue operation failed because the robot was not waterproofed and water caused it to malfunction.

In 2011, when 9.0 magnitude earthquake and tsunami hit Japan, 2,500 evacuation centers were set up to accommodate displaced and injured people. People were transported from the disaster areas to the evacuation centers. Survival of the disaster-stricken people depended a lot on planning the most efficient path. Once again, a clever shortest path algorithm was needed to find the quickest route for the rescue work.

These are examples of applications that reveal the importance of finding the best way to solve the shortest path problem. Even though the shortest path problem has been investigated since 1950s, research in this area is still active and many applications are still using it.

This thesis presents a new algorithm, which provides an alternative to the fastest algorithm that solves the all pairs shortest path problem (APSP) in $O(n^2 \log n)$ expected time. A well known Moffat-Takaoka (MT) algorithm (1) that manages to solve the APSP in the fastest time was developed almost three decades ago. Since then, there has been no other algorithm that can solve the problem better than that. The use of a critical point in the algorithm makes the algorithm hard to analyze to see the running time of the algorithm. The first contribution of this thesis is to show that a small modification of the MT algorithm can achieve the optimal complexity of $O(n^2 \log n)$ with a simpler analysis. This thesis also presents two new data structures to facilitate the process of finding the shortest distance. The first data structure, which is comparable to the trinomial heap (2), shows better performance in the total number of key comparisons, when n values are small (n values denote the number of vertices in a graph). The second data structure, a dimensional heap that is forced to maintain the thinnest structure possible has also been developed. Surprisingly, if m decrease-key operations are called (m is the number of edges in a graph), the dimensional heap shows outstanding results. Empirical studies demonstrate that this data structure is able to perform better than the existing binary, Fibonacci (3) and 2-3 heaps (4).

The chapters in this thesis are organised as follows. Chapter One briefly gives an introduction to the shortest path problem. Chapter Two provides background information for the shortest path problem, including a long history of the problem, some basic concepts that readers need to know, and previous algorithms and data structures used to solve the shortest path problem. Chapter Three explains two new data structures that have been developed with associate results obtained from the experiment. Next, Chapter Four describes a new alternative algorithm that solves the all pairs shortest path problem in $O(n^2 \log n)$ expected time. This chapter also produces some experimental results. The last chapter summaries all the findings and concludes all chapters.

2

Shortest Path Background

This chapter presents some background information about shortest path problems. The graph concept is addressed in this chapter for better understanding of the problems. Time complexity is briefly mentioned to measure the efficiency of the computation method used in this thesis. Some interesting historical achievements in research in this area are described. This includes research in the shortest path algorithms and research in the data structures, mainly in the heap structures. As Dijsktra's algorithm provides the foundation of many shortest path algorithms, detailed explanation of this algorithm is included. Floyd's algorithm is also described in this chapter as this algorithm is a pioneer algorithm in solving all pairs shortest path problems. Descriptions for a few data structures such as binary, Fibonacci, 2-3 heap and trinomial heaps are also given.

2.1 Introduction

If people are asked how to find a route from one city to another, they can easily provide a few suggestions. This is a very simple question that almost everyone can easily understand and sometimes help us find solutions. Sometimes a machine may be used to solve the problem when the input data is large. This is why the idea of the shortest path problem as a fundamental problem in the computer science area was conceptualized. When the programming pioneer Edsger Dijkstra was asked to demonstrate the performance of the new ARMAC1 machine in 1956, he had to find a problem that everyone could understand, as not everyone knew about computers at that time. To do that, he designed a program to find the shortest route between two cities in the Netherlands. The program used 64 cities in the Netherlands to demonstrate the performance of the new machine for solving the problem. Even though this great idea only took Dijkstra 20 minutes to design the algorithm (without using any pen or paper), now this idea is studied worldwide and Dijkstra's method has been used for solving many related problem.

The next section discusses basic concepts that one must know in order to understand this topic further.

2.2 Basic Terminology

The shortest path problem is represented by a graph, $G = (V, E)$ where V is the set of vertices or nodes and E is the set of edges or arcs. Sometimes, the graph is known as a network. Each single graph has its own properties which distinguishes the graph from one type to another. Different graphs also require different methods of implementation. Therefore, to find ways or algorithms to solve the shortest path problem, one should know the graph. This is mainly because the shortest path problem is solved based on the type of graph and its properties that are used.

The shortest path problem can be classified into three main problems: single pair shortest path (SPSP), single source shortest path (SSSP) and all pairs shortest path (APSP). If the problem is finding the shortest path between two locations, the problem is called SPSP. Sometimes, this problem is also called point-to-point problem. When the shortest route is sought from one vertex to all other vertices in the graph, for example, finding the shortest route from one city to all available cities, the problem is known as the SSSP problem. The APSP problem is a problem for determining the shortest path from each vertex to every other vertices in a graph.

Since Dijkstra introduced the shortest path problem, many algorithms have been developed to solve these problems. One algorithm claims that its implementation is better than another, and others demonstrate that the algorithms that they have developed can be executed faster or simpler than others. Thus, a good computational model should be used as a tool to analyze and compare algorithmic efficiency. A comparison-addition model and a word-RAM (Random Access Machine) model are two common independent methods to measure time complexity.

The comparison-addition model is briefly explained in (5). This model assumes that all input distance data consist of m real numbers. Only comparison and addition can be performed on distance data, apart from operations on control variables. For the algorithm that deploys this computational model, operations for execution should depend on the output of the comparison operation. Each addition or comparison operation is assumed to have been done in constant time. A word-RAM model, on the other hand, is a simple model of computation. Its memory consists of an unbounded sequence of registers which each hold an integer value or a real number. Memory instructions involve arithmetic operations, comparisons and bitwise boolean operations. In this model, it is assumed any register can be read or written in constant time. However, this thesis focuses only on the use of the comparison-addition model involving key comparisons.

The computational model measures the time complexity to compare the performance of one algorithm to another. The amount of time taken to solve one particular problem with the increase of input sizes is what defines time complexity of the algorithm. Sometimes, space complexity is also used to compare the efficiency of the algorithms. However, choosing which complexity is more important to use often depends on the limitations of the technology available at time of analysis. For the time complexity, the objective of this measurement is to determine the feasibility of an algorithm by estimating an upper bound on the amount of work performed.

To know more about time complexity, the next section is given.

2.2.1 Time Complexity

Time complexity, $T(n)$, is measured using Big O notation, a family member of Landau notation. The letter O is used because the rate of growth of a function is also called its order. The time complexity, $T(n)$, of an algorithm is $O(f(n))$ if, for a positive constant, C, with n input sizes,

$$T(n) \leq Cf(n) \tag{2.1}$$

In 2.1, $T(n)$ grows at the order of $f(n)$ and therefore, it can be written as $T(n) = O(f(n))$. This indicates that the running time of $T(n)$ cannot exceed the functional order of $f(n)$. The analysis of this time complexity is also known as asymptotic analysis. Common functions obtained when analyzing algorithms are shown in Table 2.1.

Notation	Name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Table 2.1: Naming functions in the big O notation

Seeing the performance of algorithms means seeing the growth rate of this function. Let two algorithms A and B solve a problem C and let $f_A(n)$ and $f_B(n)$ be the time complexities of algorithms A and B, respectively. If both algorithms have the same time complexity, that means, $f_A(n) \approx f_B(n)$. Algorithm A is said to be better if $f_A(n) = O(f_B(n))$.

In addition to the big O notations, another Landau symbol used in computing the running times is described in Table 2.2. However, as the big O notation is widely used for comparing functions, this thesis will use only big O notation for describing the running time. Note that in Table 2.2, C ia a constant variable.

Notation	Definition	Analogy
$f(n) = O(g(n))$	$f(n) \leq Cg(n)$	$\leq$
$f(n) = o(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$<$
$f(n) = \Omega(g(n))$	$g(n) = O(f(n))$	$\geq$
$f(n) = \omega(g(n))$	$g(n) = o(f(n))$	$>$
$f(n) = \theta(g(n))$	$f(n) = O(g(n))$ and $g(n) = O(f(n))$	$=$

Table 2.2: Landau notations used for describing time complexities

There are three types of time complexity analyses, which are worst-case, average-case and best-case. In the worst-case analysis, $T(n)$ = maximum time taken for an algorithm to solve the problem on any input of size n. This time complexity is commonly used in analysis as it is guaranteed that each operation requires less than $T(n)$ time to finish the task. When some assumptions must be made, for example an assumption of statistical distribution of inputs are needed, $T(n)$ = expected time of algorithm over all inputs of size n. This form of analysis is known as average-case analysis. In the average case, when the same set of operations are executed more than once, different running time are obtained. This mainly because algorithm performance depends on the type of input to the algorithm. Best-case analysis, the last type of analysis, usually works fast only on some special inputs. This analysis, however, is rarely used for comparing the performance between two algorithms.

Amortized cost analysis of running time is also used in this thesis. In amortized cost analysis, a sequence of operations is analyzed. If an operation takes $T(n)$ time and m operations are performed, worst case analysis gives the total time of $mT(n)$, rather pessimistic. If most operations take less than $T(n)$ time, the average time may become much smaller. The term "amortized" rather than "average" is used, and it is said that the amortized time is much smaller, because "average" is used when randomness comes from the input data. In the amortized analysis, there is no concept of randomness. To help in amortized analysis, the concept of potential may be used. The idea is that some expensive operations can increase the potential, so that later operations can be done cheaply, thanks to the increased potential. Detailed examples will be seen in Chapter Three.

As the shortest path problem can be represented by a graph, the next section will review some essential graph terminology for better understanding of the problem.

2.3 Graph Terminology

In the shortest path problem, the graph is used to represent data or a problem to be solved. A graph, G, is defined as a data structure that consists of a set of vertices or nodes, V, and a set of edges or arcs, E, $G = (V, E)$. A common notation used in the graph is n to denote the number of vertices, $n = |V|$ and m to denote the number of edges, $m = |E|$. An edge is defined as a pair of vertices. It can be represented by (u, v) to show that vertex u and v are connected. A directed edge may also be represented as (u, v) where u is known as the origin vertex and v is called the destination. The unordered pairs are known as undirected edges.

Edges which have associate costs are known as a weighted edges. Such weights might represent air-fare cost, the distance between two locations, the speed limit between two points and so on. The edge weight typically shows cost of traversing from one node to another node. A weighted graph is a graph that has weighted edges; otherwise, the graph is known as unweighted graph. The unweighted graph shows the existence of a connection between two nodes. To show that one operation must be done first, before another operation in job scheduling, an unweighted graph can be used.

Graphs can also be classified into directed and undirected graphs. A graph that has all directed edges is known as a *directed graph* or *digraph.* In a directed graph, each edge can only be traversed in a specific direction. The edges are drawn as arrows and can only be traversed by following the direction of said arrows. Thus, edges (u, v) and (v, u) are not the same edges. If the edges of the graph are drawn with no arrow, that means they can be traversed in either direction. This type of graph is know as an *undirected graph.*

When the number of edges, m, is close to n^2, where n is the number of vertices, a graph is said to be *dense.* An apposite for the dense graph is a *sparse* graph. A sparse graph has only a few edges, such as $m = 2n$. Outgoing edges and incoming edges of vertex x are terms used to describe the edges from and into vertex x. In a directed graph that has n vertices, the number of edges, $m \leq n(n-1)$. For an undirected graph, $m \leq \frac{n(n-1)}{2}$. Therefore, a graph with n vertices has at most $O(n^2)$ edges.

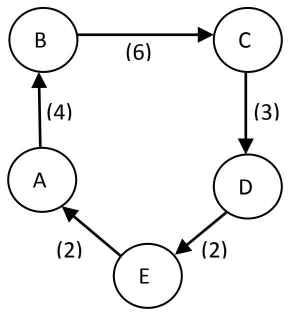


Figure 2.1: An example of a digraph

For an easy explanation, see Figure 2.1, which shows a simple digraph that has 5 vertices and 5 edges, $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (B, C), (C, D), (D, E), (E, A)\}$. The cost for edge (A, B) is 4, $cost(A, B) = 4$. The graph is a sparse graph with $m = n$.

One of the most basic graph terminology related to shortest paths is that of a path. A *path* is defined as a sequence of vertices in which each pair of successive vertices is connected by an edge. The first vertex in the path is called a start vertex or an origin vertex, while the last vertex is known as the end vertex or a destination vertex. A path, P can be written as $P = ((v_1, v_2), (v_2, v_3), \ldots, (v_k, v_{k+1}))$, where a pair $(v_i, v_{i+1}) \in E$.

Sometimes a special path is obtained from a vertex v back to itself. This path is known as a *cycle* in the graph. *acylic* is a term used to decribe a graph in which there are no cycles.

There are two common ways to represent a graph. The first technique is to use an adjacency matrix to create the graph and the second technique is to use an adjacency list. In the adjacency matrix, a graph is created by storing the adjacency information in a matrix of $|V| \times |V|$. In such a matrix, rows represent source vertices and columns represent destination vertices. Each pair is considered as an edge and the cost for this edge is stored in the matrix.

Using the adjacency list representation requires vertices to be stored as records. There is a list of adjacent vertices for each vertex $v \in V$ to show the destination vertices from v. The associate edge cost can be stored in the list structure.

In this thesis, most of the graphs used were created using the adjacency list representation. In the graphs, the edge costs were randomly generated to have random integer values.

The followings section will briefly explain past research in the area of the shortest path problem.

2.4 Shortest Path Algorithms

In the late 1950's, when the shortest path (SP) problem was established, many computer scientists tried to solve the problem. Different techniques or algorithms were proposed based on the type of problem which needed to be solved. Mostly, this depends on the weight given to the graph and the problem size. The SP problem size is measured by the number of vertices, n, and the number of edges, m in a graph. The values of n and m reflect the type of the graph, whether dense or sparse graphs.

Edge weight or cost varies from one graph to another. For a weighted graph, there is a certain edge cost assigned to each edge of the graph. Edge cost may be either negative or non-negative values. Not all algorithms that solve the shortest path problem accept all types of values. Usually, different edge cost values will require different algorithms to solve the SP problem.

The first established algorithm is known as Dijkstra's algorithm (6). This algorithm can be used to work out the shortest path problem if a graph with non-negative edge costs is given. When the algorithm was introduced, there was no priority queue used with the algorithm. The time complexity of this algorithm was $O(n^2)$. In 1984, when the Fibonacci was developed, Dijkstra's algorithm solved the single source shortest path (SSSP) problem in $O(m + n \log n)$, where n and m represent the number of vertices and edges in the graph.

When the given edge costs are in a range, for example each edge cost is bounded by $[0, C]$, better time complexity can be achieved. The idea in terms of bound C was first introduced by Dial (7) in 1969. Using the edge cost in the range $[0, C]$, Dial managed to solve the SSSP problem in $O(m + nC)$. Ahuja et al. (8) explored this idea by introducing different priority queues to be used by Dijkstra's algorithm. With the newly implemented one-level form of radix heap, they managed to solve the SSSP in $O(m+n \log C)$. When the two-level form of radix heap and the combination of a radix heap and Fibonacci heaps were used, Dijkstra's algorithm solved the SSSP in $O(m + \frac{n \log C}{\log \log C})$ and $O(m + n\sqrt{\log C})$ respectively. Later, Cherkassky, Goldberg and Silverstein (9) improved the performance to $O(m + n\sqrt[3]{\log C}^{1+\varepsilon})$ expected time for any fixed $\varepsilon > 0$. In 2003, with some improvement on certain operations in the priority queue used, Thorup (10) managed to solve the SSSP problem in $O(m + n \log \log C)$ or $O(m + n \log \log n)$.

The most recent result in the area of SSSP was discovered by Orlin et al. (11). They have shown that in a situation where only few distinct edge costs allowed, the SSSP problem can be solved in linear time. Orlin et al. in (11) also suggested that to get $O(m)$ time complexity, the number of distinct edge costs, K, should be less than the density of the graph, $nK \leq 2m$. Otherwise, the algorithm runs in $O(m \log \frac{nK}{m})$ time. To obtain required results, an efficient technique was used for implementing Dijkstra's algorithm to solve the SSSP. Even though various improved results have been discovered, Dijkstra's algorithm remains the best original technique used to resolve the SSSP problem.

Dijkstra's algorithm can also be utilized to solve the Single Pair Shortest Path (SPSP) problem. Here, the shortest path between two locations is sought. A well-known algorithm for solving the SPSP problem is known as A^* *search* algorithm (12). The A^* *search* algorithm is essentially the same as Dijktra's algorithm, except there is a heuristic concept introduced in this algorithm. In a *heuristic* concept, approximate solutions are suggested when solving the problem. The process usually estimates which is the best node to search next rather than searching all nodes, one by one. By combining the efficiency of heuristics, performance to solve a particular problem can be greatly improved. This is exactly what is needed in real-time systems such as path finding. Another commonly used algorithm is *bidirectional search* that runs two simultaneous searches (13). Later, some enhancement to the bidirectional search with heauristic approaches resulted in many new algorithms such as proposed in (14) (15) (16) (17). Performances for some of these algorithms were compared in (18).

There are many other algorithms focus on a pre-processing technique to solve the shortest path problem especially when a large network is involved. The most common technique used is dividing a graph into a number of disjunct subgraphs connected by a boundary graph, called *highway hierarchies*. The highway hierarchies approach was introduced by Sanders and Schultes (19) in 2005. They conducted experiments with a real-world road network to test the effectiveness of the approach. As a conclusion, they have suggested that the highway hierarchies method is not only promoting space efficiency, but also modest and robust (20). However, only undirected graphs were used in their system. Some improvement over this technique can be found in (21).

For the all pairs shortest path (APSP) problem, Floyd's algorithm (22) can be used. Floyd's algorithm has a time complexity of $O(n^3)$, which is equivalent to performing Dijkstra's algorithm n times. As the APSP algorithm is the main theme of this thesis, Floyd's algorithm will be discussed further in the next section.

All the shortest path algorithms discussed above only work for the non-negative edge costs. If the negative length is allowed, Bellman Ford's algorithm (23) is one of the good algorithms to choose. This algorithm runs in $O(mn)$ time in solving the SSSP problem. For the APSP problem, Johnson's algorithm (24) is the best option for the negative edge costs. Johnson's algorithm can solve APSP problem in $O(mn+n^2 \log n)$. For a sparse graph, Johnson's algorithm performs better than Floyd's algorithm as the complexity of Floyd's algorithm is $O(n^3)$.

For the sake of theoritical explanation, Cherkasky et al. (9) ran several experiments to observe the behavior of different types of shortest path algorithms. For a non-negative edge cost, Dijsktra's algorithm was found to be a robust algorithm. They also observed that a specific problem structure effected the performance of an algorithm. The performance of the algorithm also decreases when small changes such as an addition of an artificial source are added to the algorithm.

Many algorithms execute $n\times$ Dijkstra's algorithm to solve the APSP problem. However, different time bounds obtained depend on the technique used to solve the problem. When Dijkstra's algorithm is used to solve the APSP problem, the time execution is $O(mn+n^2 \log n)$. Here, Dijkstra's algorithm is used together with the Fibonacci heap.

It can be seen that two parameters are used when describing the time complexity for an algorithm. This is mainly due to the density of the graph used. For a dense graph, $m = O(n^2)$ while for a sparse graph, $m = O(n)$. Therefore, for a sparse graph, m and n parameters are used to represent the complexity of the APSP, while for a dense graph, only the n parameter is used.

The best time complexity in the area of sparse graph for the APSP algorithm was explored by Seth Pettie (25). The complexity achieved was $O(mn + n^2 \log\log n)$. This complexity beat the long-standing complexity of $O(mn + n^2 \log n)$ which uses Dijsktra's algorithm with the Fibonacci heap implementation.

In the area of dense digraphs, complexity is measured using two analyses, worst case and average case analysis. The best known result for the worst case is slightly sub-cubic by Han and Takaoka(26), which is in $O(\frac{n^3(\log\log n)}{\log^2 n})$. In (27), Chan summaries all good achievement APSP algorithms for general dense real-weighted graphs. Readers are advised to see Table 1 in (27) for the summary. The other area is for the average case analysis, which is the main theme of this thesis. Algorithms that solve APSP in the expected time analysis will be discussed in detail in Chapter Four.

The following sections discuss two algorithms that are commonly used to solve the SP problem.

2.4.1 Dijkstra's Algorithm

Given a weighted graph, $G = (V, E)$, where V is a set of vertices and E is a set of edges, and s is a source vertex in V, find the path of shortest length connecting s to all vertices in V. This is the SSSP. The problem is trying to get the minimum cost between vertex s to all other vertices, $v \in V$. If the minimum cost is found, then $d(v)$ denotes the minimum edge cost from s to v. Initially, $d(s)$ is assumed to be 0, $d(s) = 0$ to denote that s is the source or the start vertex.

The length of the directed edge connecting vertex u to vertex v is represented as $cost(u, v)$.

The eponymous Dijkstra's algorithm is used to solve the single source shortest path problem (SSSP) for a non-negative edge lengths graph. The single source shortest path algorithm is described in the following. Let $G = (V, E)$ be a directed graph where V is the set of vertices and $E \subseteq V$ x V is the set of edges. $OUT(v)$ is defined as a set of vertices w such that there is a directed edge from vertex v to vertex w. The non-negative cost of edge (v, w) is denoted by $cost(v, w)$. It is assumed that $cost(v, v) = 0$ and $cost(v, w) = \infty$ if there is no edge from v to w. A vertex s is denoted as the source vertex. The shortest path from s to vertex v is the path such that the sum of edge costs of the path is minimum among all paths from s to v. The minimum cost is also called the shortest distance. In Dijkstra's algorithm, two set of vertices, S and F, are maintained. The set S, called the solution set, is the set of vertices to which the shortest distances have been finalized and the set F, called the frontier, is the set of vertices which can be reached from S by a single edge. Vertices that remain outside S and F are considered unexplored vertices that need to be explored.

In solving a single-shortest path problem, Dijkstra's algorithm maintains a distance value $d[v]$ for each vertex v in the graph. The value of $d[v]$ indicates the shortest distance from the source vertex to vertex v. If v is in F, $d[v]$ is the distance of the shortest path that lies in S except for the end point v.

Initially, the source vertex s, with $d[s] = 0$ is put in S. Vertices in $OUT(s)$ are put in F with their keys values. The key value of $v \in OUT(s)$ is computed as $d[v] = d[u]+$ cost(u, v). The algorithm works as the following:

1. A vertex v that has the minimum distance among those in F is selected.

2. If v is outside S, move v from F to S and the following steps are taken. Otherwise, the first step is repeated.
3. The shortest distance from s to v, $d[v]$ is now known and finalised.
4. For every vertex $w \in OUT(v)$, a new distance key, *key*, is calculated by adding the shortest distance of v and the edge length from v to w, $key = d[v] + cost(v, w)$.
5. If w is already in F, the new *key* is compared with the existing $d[w]$ and the minimum distance is assigned to $d[w]$.
6. If w is not in F, it will added into F with $d(w) = key$.

This process continues from the first step until there is no vertex in F. If F is empty, the shortest distance from s to all vertices has been finalised and all vertices are now known as labelled vertices. Algorithm 1 shows Dijkstra's algorithm to solve the shortest path problem.

Algorithm 1 Dijkstra's Algorithm

```
$\forall v \in V : d[v] = \infty$;
$S = \varnothing$; $d[s] = 0; F = \{s\}$;
while $|S| < n$ do
    find $v$ in $F$ with $d[v] = \min\{d[i] : i \in F\}$; $/ * delete - min * /$
    $S = S + \{v\}; F = F - \{v\}$;
    for each vertex $w \in OUT(v)$ do
        if $w \notin S$ then
            if $w \in F$ then
                if $d[v] + cost(v, w) < d[w]$ then
                    $d[w] = d[v] + cost(v, w)$; $/ * decrease - key * /$
            else
                $d[w] = d[v] + cost(v, w)$; $/ * insert * /$
                $F = F + \{w\}$;
```

There are a few operations involved in running the Dijkstra's algorithm. When a vertex with the minimum key value is removed from F, a *delete-min* operation is called. When a vertex with certain key value needs to be updated in F, a *decrease-key* operation is used. Inserting a new vertex in F requires an *insert* operation. Therefore, a good data structure that supports theses operations is needed when executing Dijkstra's algorithm. The data structure used to support the operations should have a reasonable time complexity. In a graph that has n vertices and m edges, delete-min and insert operations are executed n times. The decrease-key operation is done in m times in the worst case. As the decrease-key operation is done very frequently, especially when $m \approx n^2$, many kinds of data structures, such as binary (28), Fibonacci (3), 2-3 heaps (4) and trinomial (2) heaps try to reduce the decrease-key complexity by adding complexity on the delete-min function.

Analysis of Dijkstra's algorithm depends on the data structure used. If the data structure is a linear array, the total running time of the algorithm is $O(m + n^2)$, that is $\approx O(n^2)$ time, where $n = |V|$ and $m = |E|$. This is because the delete-min takes $O(n)$ time and it has to be performed n times. Therefore a total time for the delete-min is $O(n^2)$. Each operation of decrease-key takes $O(1)$ time and there are m operations. Hence, the running time of decrease-key operation is $O(m)$ time. To run Dijkstra's algorithm using a linear array data structure will take $O(m + n^2) \approx O(n^2)$ time.

If the Dijkstra algorithm is run using a binary heap data structure, the running time is different from the above. In the binary heap, delete-min takes $O(\log n)$ time. The insert and the decrease-key operations take $O(\log n)$ time. Hence, the total running time of the algorithm with the binary heap is $O(m \log n + n \log n) = O((m + n) \log n)$ time. If all nodes are reachable from the source, s, then the running time becomes $O(m \log n)$ time. For higher graph densities, the number of edges, m is almost n^2; this will give a time of complexity $O(n^2 \log n)$ time if Dijkstra's algorithm is running with the binary heap.

Table 2.3 shows how Dijkstra's algorithm works when a graph in Figure 1.2 is used. The solution set is labelled as **S** and $d[i]$ shows distance of the shortest path.

Iteration	**S**	$d[1]$	$d[2]$	$d[3]$	$d[4]$	$d[5]$
1	{0}	3	5	∞	∞	∞
2	{0, 1}	3	5	4	∞	∞
3	{0,1, 3}	3	5	4	8	6
4	{0,1 , 3, 2}	3	5	4	7	6
5	{0, 1, 3, 2, 5}	3	5	4	7	6
6	{0, 1, 3, 2, 5, 4}	3	5	4	7	6

Table 2.3: Performing Dijkstra's algorithm on Figure 1.2

When $n\times$ SSSP problem is solved, the APSP problem is catered. Many APSP algorithms that have been developed employ this method to solve the problem. The common APSP algorithm is Floyd's algorithm that will be discussed in the following section. This algorithm, however use matrix operation rather than using the common technique, $n\times$ SSSP algorithm in solving the APSP problem.

2.4.2 Floyd-Warshall Algorithm

Floyd's algorithm (22) is designed to find the shortest path between all the vertices in a graph. To implement this algorithm, a two-dimensional array is usually used to build a matrix. The size of the matrix is $n \times n$, where n represents the total number of vertices in a graph. Each row in the matrix denotes a "starting" vertex in a graph while each column in the matrix represents an "ending" point in the graph. If there is an edge that connect a starting vertex i and an ending vertex j, then the cost of this edge is placed in position (i, j) of the matrix with the edge cost value, $d[i, j]$. The edge cost 0 is given when the stating vertex and the ending vertex are equivalent, i.e. $d[i, i] = 0$. An infinite value is placed in the (i, j) position of the matrix if there is no edge that connect the two vertices, i and j. This is to specify the impossiblity of directly moving from i to j.

The fundamental basis of Floyd's algorithm is to determine whether a path from a vertex i to j via k is shorter than the existing best known path from i to j, $d[i, j]$. The new path is given as $d[i, j] = d[i, k] + d[j, k]$. If the new path that goes via k is shorter, then, the old value of $d[i, j]$ will be replaced with the new value. Figure 2.2 shows simple paths from vertex i to vertex j.

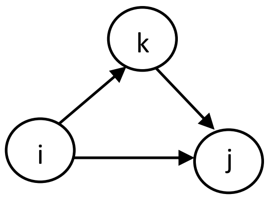


Figure 2.2: An example of a path from i to j via k.

A detailed description Floyd's algorithm is given below.

Algorithm 2 Floyd's Algorithm

```
1: if there is no edge between $i$ and $j$ then
2:     $d[i,j] = \infty$;
3: else
4:     $d[i,j] = cost(i,j)$;
5: if $i$ is equal to $j$ then
6:     $d[i,j] = 0$;
7: for $k = 1$ to $n$ do
8:     for $i = 1$ to $n$ do
9:         for $j = 1$ to $n$ do
10:            if $d[i,j] > d[i,k] + d[k,j]$ then
11:                $d[i,j] = d[i,k] + d[k,j]$;
```

From the Algorithm 2 above, it is clear that Floyd's algorithm takes $O(n^3)$ time to execute the APSP. The time complexity obtained in solving APSP by Floyd's algorithm is a worst case time. However, this thesis mainly focuses on analyzing the APSP algorithm using an average case analysis. Some algorithms that solve the APSP problem in the average case time are Spira (29), Bloniarz's algorithm (30) and Moffat-Takaoka (MT) algorithm (1). The details of these algorithm are explained in chapter 4.

The next section will discuss the heap data structures and explains the commonly used data structures such as binary, Fibonacci, 2-3 heap and trinomial heaps.

2.5 Shortest Path Data Structures

When solving the shortest path problem, the use of a good data structure is essential. Data structures work closely to serve algorithms, in order to improve the running time to solve the problem. In the worst case running time, algorithm performances can be improved by clever data structures. Consider a few cases below that show the effect of using data structure to the algorithm performances.

Dijkstra's algorithm When a priority queue is used, Dijkstra's algorithm can solve the shortest path problem in $O(m + n \log n)$ time. Without the priority queue, the best time for solving the problem by Dijkstra's algorithm is in $O(n^2)$.

The maxflow algorithms: When dynamic trees are used by the maxflow algorithms, time complexity is improved from $O(n^2\sqrt{m})$ time to $O(nm)$ time.

Algorithm for general weighted matchings: With the use of mergeable priority queues, an algorithm for general weighted matchings is able to solve the problem from $O(n^3)$ to $O(nm \log n)$.

The above cases show that it is very important to use data structures for solving a particular problem, as the time complexity will change drastically. Different types of data structures used to solve a specific problem also may have different running times.

When Dijsktra's algorithm solved the shortest path problem using Fibonacci heap, it showed that SSSP problem could be solved in $O(m + n \log n)$ time. However, when other heaps were used to replace Fibonacci heap, different time complexities were obtained. In 1990, when radix heap (8) was introduced to use with Dijkstra's algorithm, time complexity obtained was $O(m + n \log C)$ where C was the maximum edge cost. When some modifications were made to the radix heap with two-level form, the complexity of executing Dijkstra's algorithm became $O(m + \frac{n \log C}{\log \log C})$. Furthermore, when radix heap was combined with Fibonacci heap to solve Dijkstra's algorithm, the performance obtained was $O(m+n\sqrt{\log C})$. The complexities obtained show that using different data structures resulted in different runtime.

Open problem by (8) whether SSSP could be solved in $O(m+n \log \log C)$ has been answered by (10). Mikkel Thorup in (10) shows that with the integer priority queue that performs decrease-key operation in constant time, the SSSP problem can be solved efficiently, that is in $O(m + n \log \log C)$ time.

The data structures used in solving the shortest path problem are classified into two categories depending on the type of analysis used. The first one is data structures with the worst case analysis and the second one is the data structure with the amortized cost analysis.

In worst case analysis, a well known binary heap (28) is the first heap that should be highlighted. Almost in every application that requires a priority to be used, this heap is chosen for its simplicity and ease of implementation. The heap is also stable and manage to perform well. For main operations such as insert, delete-min and decrease-key operations, this heap performs all operations in $O(\log n)$ time. Following the binary heap is a leftist heap, developed by Crane (31) in 1972. Then, binomial heap (32) was developed by Vullemin that also supports all the heap operations in $O(\log n)$ worst-case time per operation.

Leading amortized cost analysis is Fibonacci heap, which was introduced in 1987 by Fredman and Tarjan (3). In this heap, insert and decrease-key operation are done in $O(1)$ amortized time and delete-min in $O(\log n)$ amortized time. However, this heap has its limitations. As a practical matter, this heap is not efficient; it is also hard to implement, as the structure of this heap is complicated (33)(34)(35).

The skew heap (36) that allows self-adjusting structure was developed by Sleator and Tarjan. This heap is the amortized version of leftist heap and has the same complexity as Fibonacci heap. However, decrease-key is performed in $O(\log n)$ time. Driscoll, Gabow, Sharairman and Tarjan therefore introduced a relaxed heap (37) in 1988. This is the first heap that allows the heap order to be violated. That means that the key value of a child node is allowed to be smaller than the parent's key value. The relaxed heap uses the same concept as the binomial heap. This heap gives theoretical improvement over Fibonacci heap for the achievement in the worst case analysis. However, the heap is also difficult to implement.

A new heap, called a 2-3 heap, was introduced in 1999 by Takaoka (4). This heap uses the idea of 2-3 tree. Using dimension and workspace structure to design the decrease-key operation, this heap practically performs better than Fibonacci heap. A year later, Takaoka introduced trinomial heap (2) that supports the decrease-key operation in $O(1)$ worst case time. This heap employs the idea of a bad child or a violation node introduced in the relaxed heap (37). Compared to a relaxed heap that uses binary linking, a trinomial heap applies ternary linking in its implementation.

A pairing heap, developed by Fredman, Sedgewick, Sleator and Tarjan, is another efficient heap in practice (34). With the self-adjusting structure, the objective of introducing this heap was to beat the performance of Fibonacci heap. The heap is based on the binomial heap, but it was developed in the amortized time. However, the amortized cost from the decrease-key is not constant. It is difficult to analyze the decrease-key operation of this heap. Firstly, Fredman provided analysis for the decrease key operation as $\Omega(\log\log n)$ (33). This analysis, however was reviewed by Pettie (38), and he proved that the decrease-key operation was done in $O(2^{2\sqrt{\log\log n}})$. A small modification was made to the pairing heap; with the modification, Elmasry (39) gave $O(\log\log n)$ for the decrease-key operation.

Other heaps that have the same complexity as Fibonacci heap are thin and thick heaps (40) and quake heap (41). Heaps recently developed include the rank-pairing heap (42), violation heap (35) and strict heap (34).

With any other data structures such as Fibonacci, 2-3 heaps or trinomial heap, the running time depends on how the operations of delete-min and decrease-key are performed. For example, if the delete-min takes $O(\log n)$ time and decrease-key is in $O(1)$ time, then the total running time is $O(m + n \log n)$ time. Note that the expected number of decrease-key operations when solving the shortest path problem is $O(n \log(\frac{m}{n}))$ (43).

Table 2.4 summarizes some famous figures and their invented data structures.

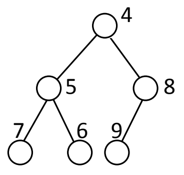
Year	Author	Heaps
1964	William	Binary heap
1972	Crane	Leftist heap
1978	Vuillemin	Binomial heap
1986	Sleator and Tarjan	Skew heap
1987	Fredman and Tarjan	Fibonacci heap
1988	Driscoll, Gabow, Sharairman and Tarjan	Relaxed-heap
1990	Ahuja, Mehlhorn, Orlin and Tarjan	Radix heap
1999	Takaoka	2-3 heap
1999	Fredman, Sedgewick, Sleator and Tarjan	Pairing heap
2000	Takaoka	Trinomial heap
2008	Kaplan and Tarjan	Thin and thick heaps
2009	Chan	Quake heap
2010	Elmasry	Violation heap
2012	Brodal, Lagogiannis and Tarjan	Strict Fibonacci heap

Table 2.4: The great data structures inventors

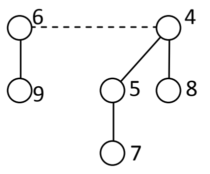
The next section will describe four types of priority queues or heaps used in solving the shortest path problem. In the explanation, three important operations will be discussed. They are insert, delete-min and decrease-key operations as these are the main operations required for solving the shortest path problem. Insert operation is a process to insert a new node or element into the heap. The delete-min process removes a node that has the minimum key value from the heap, while the decrease-key decreases the key value of a node to a new lower key value.

The heap structure varies from one priority queue to another. Let the following be an example of a sequence of operations:

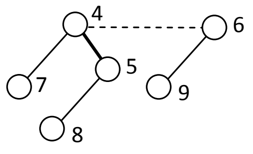
```
1: insert(5);
2: insert(3);
3: insert(8);
4: insert(2);
5: delete-min();
6: insert(7);
7: insert(4);
8: delete-min();
9: insert(1);
```



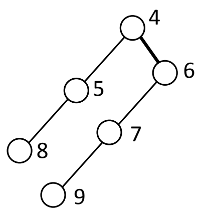
(a) Binary heap



(b) Fibonacci heap



(c) 2-3 heap



(d) Trinomial heap

Figure 2.3: Different heap structures after performing a few heap operations

```
10: insert(9);
11: insert(6);
12: delete-min();
```

The final structure of different heaps are shown in Figure 2.3 when the above operations are executed.

Brief explanations about these heaps are given in the following section.

2.5.1 Binary heap

A binary heap data structure is a complete binary tree. In this heap, each node has a higher priority than its children. The heap must be a complete binary tree. Therefore, no level is allowed to have less than two nodes except the lowest tree level. If a new node is inserted into the heap, the node will be added at the lowest level from left to right. If the key value of the inserted node is less than the parent's key value, the node will propagate to the higher level. For this insertion process, time complexity is obviously in $O(\log n)$.

Deleting a node from the heap will remove the root node as the root node has the smallest key value. When the root node is removed, binary heap requires the root's position to be filled with other nodes. To do this, a child with a smaller key value will move up to the root's position and the old position of this node will be replaced by its child node that has a lower key value. This process is repeated until one position at the bottom level heap is empty. This position will be filled up by the rightmost node in the lowest level, which is the last node in the heap. Delete-min process in the binary heap also requires $O(\log n)$.

For the decrease-key operation, the node with the lower key value after the decrease-key operation must find a new position by percolating up. When the correct position is found, the process will end. This process, therefore will take $O(\log n)$ time as well.

2.5.2 Fibonacci heap

Fibonacci heap was introduced with amortized cost complexity. This heap performs all operations such as insert and decrease-key operations in $O(1)$ amortized cost. Only delete-min operation is done in $O(\log n)$ amortized time.

The Fibonacci heap uses a collection of heap-ordered trees. Each of the tree has its root node and the root nodes are linked to each other but unordered. There is a pointer that always points to the minimum node in the heap. When a new node is inserted into the heap, the node will be placed at the root level. The node is also the only node in the new tree resulted in the insertion process. If only the insertion process is done, Fibonacci heap will have many trees and each tree has only one node. This is a relaxed structure introduced in the Fibonacci heap to maintain $O(1)$ insertion process. Fibonacci uses a degree as a term to differentiate one tree to another. The degree is defined as the number of children it has.

Removing a node from the heap is the most complicated operation in this heap. When a the minimum node is removed, the children nodes will be broken apart into smaller sub trees. These trees will be added back to the root list. Nodes in the root list that have the same degree will be merged resulting in a new structure with a higher tree degree.

In the decrease-key operation, when the old key is replaced with a new one, the heap order has to be checked. If the heap order is violated, the link between the node and its parent is truncated. The decreased node and its subtrees will be merged to the root level. At the root level, the new key of the decreased node is compared with the current minimum and the the pointer that points to the minimum node will be updated if necessary.

2.5.3 2-3 heap

The 2-3 heap shares almost similar structure to the Fibonacci heap. The trunk concept is introduced in this heap to show the number of nodes allowed in each trunk. In the 2-3 heaps, the trunk can be either 2 or 3 nodes in length. The 2-3 heap also consists of a collection of trees in different degrees. These trees are linked to each other. The 2-3 heaps defines dimension and workspace that help in performing certain operations. The lowest dimension is said to be in dim 0.

An insert operation involves merging the new node into the right most tree in the heap. During the insertion process, the heap order must be maintained. When merging the new node into the tree in the heap, it may be possible to create a carry tree. The carry key is created when the length of the main trunk is 3. The degree of the carry tree is higher by one. Thus, the carry tree will propagate to the left and merge with the same tree degree. In the worst case, the insertion process will take $O(\log n)$ as the result of the propagation.

The delete-min operation in this heap is similar to the delete-min operation in the Fibonacci heap. The link between a parent and the children nodes is removed. The minimum node is deleted from the heap while the children nodes will be merged back into the heap. In the decrease-key operation, the node whose key was decreased is first removed from the tree. This node however will be merged back to the heap at the main trunk level. When removing the node, some rearrangement of the workspace is needed. Detailed definitions and explanations of the operation are explained in (4).

2.5.4 Trinomial heap

The trinomial heap uses the idea of a bad child or inconsistent node, where a limited number of inconsistent nodes are allowed to be in the tree. In this thesis the words "inconsistent" and "active" are used interchangeably for convenience. Precisely speaking, active encompasses inconsistent. A node can become inconsistent and then consistent passively by some operation at the parent. It is expensive to check an active node is consistent with its parent. Definition of this heap is similar to the 2-3 heap. A very simple trinomial heap structure is given in Figure 2.4. In Figure 2.4, node a has two children b and c. Node b and c are called partner nodes, and normally sorted in non-decreasing order, unless they are active. Nodes b and d are called siblings. Sibling nodes are distinguished by dimensions. Node d is a child of a. Node d is a higher dimensional child of a than b is. A trunk that has nodes a and d is called the main trunk in this tree. In general the trunk of the highest dimension is the main trunk. Node b is an active node, where a black circle is used to represent this node in the tree. Node b is called the first child and node c denotes as the second child on the trunk that connects a, b and c. The first child and second child terms are used to describe the nodes's position on the trunk.

An active node is not allowed in the main trunk; if the wrong order occurs, the two nodes are swapped together with the underlying trees.

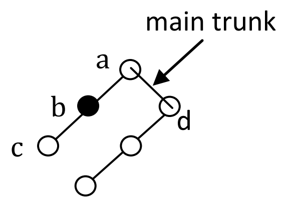


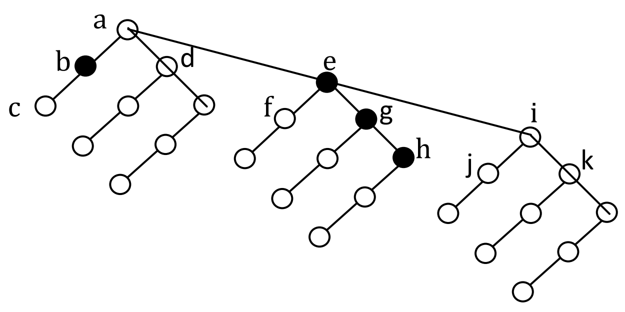
Figure 2.4: An example of a trinomial tree structure

In this heap, two internal operations are introduced. These operations are called reordering and rearrangement. Sometimes, reordering is called a promotion process. When a trunk has inconsistent nodes, that means the key value of the head node might be lower than one or two nodes that are located on the same trunk. If this happens, reordering will reorder the position of the nodes to maintain the heap property. With this technique, the number of inconsistent nodes will be reduced.

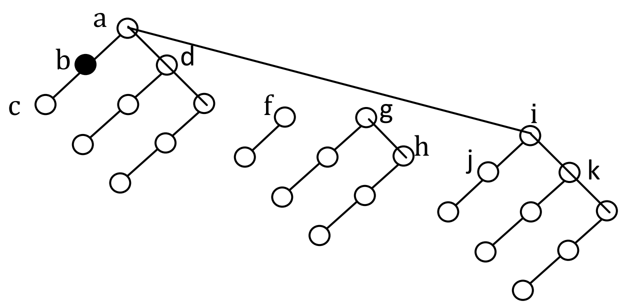
Another internal operation is rearrangement. Rearrangement is a process for rearranging nodes that are located on two different trunks. This is done when there are two trunks of the same dimensions have inconsistent nodes. Each might have one inconsistent node. The rearrangement process will rearrange the position of all nodes in these two trunks so that the heap property is maintained. Rearranging the nodes by its key values will help in maintaining the heap structure, thus reducing the number of inconsistent nodes.

The insertion process works in a similar manner to the 2-3 heap. When a new node is inserted into the heap, the node will be merged to the right most tree in the heap. If the results of the merge operation is a carry tree, the carry tree will propagate to the left. The delete-min operation is quite tricky in the trinomial heap. This is mainly because the trinomial heap allows the heap to have inconsistent nodes. To search for the minimum node means, not only the root nodes are scanned, but also the inconsistent nodes list should also be examined. This takes $O(\log n)$ time to search for the minimum node as the number of active nodes is bounded by $O(\log n)$.

During the deletion process, the number of active nodes may decrease. If the minimum node was an active node, it must be made active. This is to make sure that the number of active



(a) A trinomial tree structure before the break-up operation



(b) The tree during the break-up operation

Figure 2.5: The resulting sub trees when node e is removed from the tree. Break-up operation will result in sub trees rooted at a, f and g

nodes is decreased at least by one when the minimum node is removed.

When the minimum node is chosen to be deleted, operation *break-up* must be performed. The resulting break-up depends on whether or not the minimum node is a root node or an active node. In the break-up operation, the higher dimension parts of the tree will be broken apart, producing trunks to be merged back into the heap at the root level. The child trunk will also be merged to the root level after the link between the minimum node and the child trunk is broken. Nodes in the break-up may go from active to inactive. If the first child is an active node, the second child may be possible an active node as well. However, if the first child is not an active node, the second child activeness does not have to be checked. After the break up, the length of the main trunk decreases by one. The current tree position will become empty unless the minimum node has a partner node.

Figure 2.5 shows an example of break-up operation at node e. When e is removed from the tree, the length of the main trunk becomes 2, where nodes a and i are treated as the first and the second child node on the main trunk. Node a is also considered as a new partner of node i and the same the other way around. As a child node of the removed node, node g becomes a nonactive node. The second node on the trunk that connected nodes e, g, h, that is node h is also made nonactive. Thus, with the break-up operation, the number of active nodes is decreased by three; one from the removed node, e and another two from g and g. The new trees, rooted at a, f and g, will be merged back to the root level with the existing trees in the heap. The delete-min operation is obviously done in $O(\log n)$ time.

The decrease-key operation is quite relax in the trinomial heap. The position of the node which the key is decreased on a trunk is important. The decreased node might have to be swapped with a higher location node on the same trunk if the new key is smaller than the higher node. This is to ensure that the heap order is correct. The decreased node is made a new inconsistent node if the number of inconsistent nodes in the heap is still in tolerance. Otherwise, reordering and rearrangement process has to be performed to keep the number of inconsistent nodes under control. To describe the decrease-key process, Algorithm 3 is given. Let v be the node that the key value is decreased.

Algorithm 3 Decrease-key procedure in the trinomial heap

```
1: if $v$ is a root node OR $v$ is an active node then
2:     rearrangement is not necessary;
3: if $v$ has a parent node AND $v$ is the second child then
4:     if $key(v) < key$(first child) then
5:         swap $v$ with the first child to maintain the correct ordering;
6:     else
7:         if the first child is active then
8:             make $v$ as a new active node;
9:             if the number of active nodes reaches its limit then
10:                rearrangement is needed;
11: else                                                   ▷ $v$ is the first child
12:     $v$ will be made active ;
13:     if the number of active nodes reaches its limit then
14:         rearrangement is needed;
15: if $key(v) < key$(first child) then          ▷ $v$ is the second node on the main trunk
16:     swap $v$ with the first child to maintain the correct ordering;
```

In (2), the decrease-key operation can be implemented in $O(1)$ for both worst case and amortized time.

2.6 Shortest Path Application

The shortest path problems are common problems that are relatable to our daily life. The simple examples given in the introductory chapter was about using a Gobal Positioning System (GPS) system to find the route from Christchurch to Twizel. The second application was about robots used in the emergency procedure, and the last was about allocating displaced and injured people in the minimum time possible during the disaster.

The above applications are closely related to the route finding, which is the main application of the shortest path problem. Route finding is very important not only in the transportation system, but also in diverse areas such as in computer games engines, social networking systems as well as in the operational research.

Finding the best route to drive from one location to another is the main objective of a good transportation system. Here, the shortest path algorithm may also be used as a planning tool such as to predict the traffic flows that can be helped to find the fewest route possible during an emergency. It should also be able to provide a drive guiding system. In the computer games, the path finding is essential for the game engine to assist users in plotting routes. While, in the social networking, path finding is used to find the connection between two users.

Other shortest path applications are widely used in the operational research such as in the fleet management system in underground mines. It can also be used in routing telecommunication messages, maps and so on. Indeed, it should be used in any application where the optimal routings must be found.

3

New Data Structures

This chapter presents new data structures which have been developed to facilitate the process of finding the shortest paths by an algorithm. First, as an alternative to the existing heaps (as described in the previous chapter), this chapter will describe and formalize the new heaps, and discuss their performance and whether they are better than that of the existing ones. It focuses two types of heap structures, dense and thin. In order to determine whether the dense data structure is good, the quaternary heap has been developed. This data structure, comparable to the trinomial heap, shows better performance in the total number of key comparisons when n values are small (n denotes the number of vertices in a graph). Next, this chapter will discuss the development of a new data structure called a dimensional heap. A dimensional heap is forced to maintain the thinnest structure possible. Surprisingly, if m decrease-key operations are called (m is the number of edges in a graph), the dimensional heap shows outstanding results. Empirical studies demonstrate that this data structure performs better than existing binary, Fibonacci and 2-3 heaps.

3.1 Introduction

In this section, descriptions of tree and r-ary tree are given. A *tree* is defined as a priority queue that consists of nodes and branches. Each *branch* connects two nodes together. If each node in the tree is arranged according to its key value, this special type of tree is known as a *heap.* In the minimum heap data structure, the key value of a parent node is always lower than or equal to those of children nodes; for all nodes $v \in V$, excluding the root, $key(parent(v)) \leq key(v)$ (V denotes the set of vertices or nodes). The same concept is applied across the heap. Therefore, the root node of the minimum heap always has a minimum key value among other key values.

The root node is also the only node in the heap when there is only one node in the heap. The term heap and tree are used interchangeably to describe the heap data structure in this Chapter.

The first branch exists from the root node when a second node is added to the tree. In other words, whenever a new node is inserted to the tree, there will be a presence of a new branch connecting the existing node and the new node. A path that connects several nodes is called a *trunk*. The length of a trunk is the number of branches plus 1, that is, the number of nodes comprising the trunk. The first node in the trunk is called the *head* or *parent node*. Other nodes are *children* of the head nodes. The nodes on the same trunk are called *partners*. A tree is created when a group of trunks are connected in some fashion. An example of a tree with its terminology is given in Figure3.1.

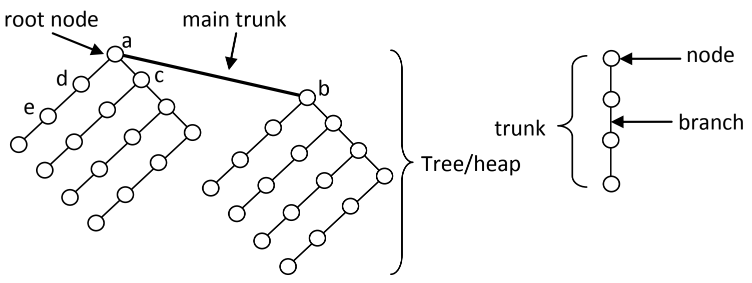


Figure 3.1: Basic terminology of a tree

Each node in a tree is said to be in certain *dimension*. Dimension of node v is stated as $dim(v)$. Let us define the dimension using Figure 3.1. If node v is located on a trunk in the lowest level (the trunk slopes towards bottom-left), it is said that node v is in dimension 1, $dim(v) = 1$. When the trunk slopes towards the bottom-right, nodes on this trunk are said to be in second dimension, $dim(v) = 2$. The parent node on each trunk is always in one dimension higher than the highest dimension child. For example, if the highest dimension child is i, then the parent node will be in dimension $i + 1$. The parent node can be the root node if it has the smallest key value. Thus, the dimension of the tree is said to be in the same dimension of the root node. In Figure 3.1, node a is the root node. Nodes c and d are children of node a. Dimensions of node d, $dim(d) = 1$ and node c, $dim(c) = 2$. The highest dimension of the child node is 2; therefore, the dimension of the root node, $dim(a) = 2 + 1 = 3$.

Two nodes are called partner nodes if they are connected to each other on the same trunk. However, they can be only classified as the partner nodes if they are in the same dimensions.

In Figure 3.1, nodes d and e are called partner nodes, and normally sorted in non-decreasing order, unless they are active.

On each trunk, the maximum number of nodes, i.e., the length is limited. This limitation depends on the type of the heap. In Figure 3.1, the length of the main trunk is 2 and that of all other is 4. The main trunk is defined as a trunk that connects two trees of the same degrees together.

The *degree* of a node is known by calculating the number of children nodes. The degree of node x is denoted as $deg(x)$. If the degree of the root node is i, then the degree of a tree is also i as it depends on the degree of the root node. In Figure 3.1, $deg(d) = deg(e) = 0$ as both d and e have no child node. As c has only one child, then $deg(c) = 1$. The degree values of $deg(a) = 2$ and $deg(b) = 2$ as each has two children nodes. The structure of the higher degree tree is always comprised of a few lower degree tree structures in it.

3.1.1 Polynomial of trees

This section provides a more formal description of a polynomial tree. Definition of the polynomial tree here is borrowed from (4) and (2). A linear list of r nodes creates a linear tree of size r. The linear tree of size r is called an $\boldsymbol{r}$ tree. Let S and T be two trees. The product of the two trees, $P = ST$, is defined in such a way that every node in S is replaced by a copy of T and every branch in S connecting two nodes u and v now connects the roots of the roots of the trees substituted for u and v in S. In general $ST \neq TS$. The sum of two trees, $S + T$ is defined as a collection of two trees S and T.

In some situations, $\mathbf{r}$ trees can be linked to each other. The process of linking these trees is known as an r-ary linking. The result of the r-ary linking is an r-ary polynomial of trees. In the r-ary polynomial of trees, there will be a collection of $\mathbf{r}$ trees. The collection of trees is said to be the sum of $\mathbf{r}$ trees.

The r-ary polynomial of trees, P, of degree $k-1$ is defined by:

$$P = \mathbf{a}_{k-1}\mathbf{r}^{k-1} + \ldots + \mathbf{a}_1\mathbf{r}^1 + \mathbf{a}_0 \tag{3.1}$$

where the size of $\mathbf{a}_i$ is $0 \leq \mathbf{a}_i \leq r-1$. In this notation, boldface for a tree and non-boldface for the size of the corresponding tree are used. The term $\mathbf{a}_i$ represents a coefficient in the polynomial while $\mathbf{r}^i$ denotes a complete r-ary tree of degree i. The coefficient in the polynomial can be calculated by counting the number of trees of size a_i. These trees are located on the main trunk.

The plus symbol,"+" in the equation 3.1 denotes the addition of a collection of trees. Bold $\mathbf{r}$ is used to describe a linear tree of size r. The different value of $\mathbf{r}$ distinguishes the type of the r-ary tree. If $\mathbf{r} = 3$, the r-ary tree is called a trinomial tree (2). The tree is defined as a quaternary tree if $\mathbf{r} = 4$ (a quaternary heap is a new heap that will be described further in the next section).

Figure 3.2 shows an example of a complete polynomial tree of degree 2. The leftmost tree in this polynomial tree has $\mathbf{a}_i = 2$ and $\mathbf{r}^2$. Thus, combining $\mathbf{a}_i$ and $\mathbf{r}^2$, this tree is represented by $\mathbf{2} \times \mathbf{r}^2$ and the node count of this tree is $2 \times 4^2 = 32$. The polynomial tree in this example is expressed as:

$$P = \mathbf{a}_2\mathbf{r}^2 + \mathbf{a}_1\mathbf{r} + \mathbf{a}_0 \tag{3.2}$$

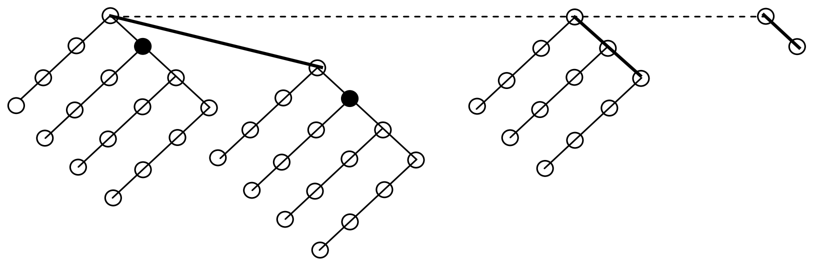


Figure 3.2: A complete polynomial tree of degree 2

In the r-ary polynomial tree, the rightmost tree is the lowest degree tree. Moving from the right to the left in the r-ary polynomial tree, the higher degree trees can be found. The right most tree in the above P can be written as $\mathbf{a}_0\mathbf{r}^0$. This tree is called $\mathbf{T}(0)$ as the degree value is 0. The tree with the degree $k-1$ is called $\mathbf{T}(k-1)$.

Then the above polynomial tree can be written as:

$$P = \mathbf{a}_{k-1}\mathbf{T}(k-1) + \ldots + \mathbf{a}_1\mathbf{T}(1) + \mathbf{a}_0\mathbf{T}(0)$$

Two trees of the same degrees can be merged by adding their coefficient values. The result of merging $\mathbf{a}_i\mathbf{T}(i) + \mathbf{a}'_i\mathbf{T}(i)$, where "+" here means the merge process, is $(\mathbf{a}_i + \mathbf{a}'_i)\mathbf{T}(i)$ if $a_i + a'_i < r$. Otherwise, the merge operation will create a carry tree $\mathbf{r}^{i+1}$. Figure 3.3 shows the results obtained when two trees are merged. Figures 3.3(a) and 3.3(b) show example of results obtained when the coefficient values, $a_i + a'_i < r$. When $a_i + a'_i = r$, the merge process will create a carry tree of one degree higher as shown in Figure 3.3(c). Figure 3.3(d) shows not only a carry tree, but also the tree with the existing degree is maintained when $a_i + a'_i > r$.

Details explanation of r-ary tree can be found in (4) and (2).

3.2 A Quaternary Heap

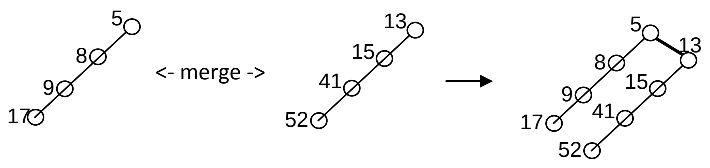
The first data structure which has been developed is called a quaternary heap. A *quaternary heap* is defined as an extended version of the trinomial heap (2). The only difference between these two heaps is the trunk size or the length of the trunk. In this quaternary heap, the length of main trunks can be 0 to 3, and other trunks have length 4. Key values in a trunk are sorted in non-decreasing order, except for the head node.

The design of a quaternary heap is also similar to a trinomial heap. When linking nodes by a trunk, a certain number of inconsistent nodes is allowed. The *inconsistent nodes* are nodes that have key values greater than the parent's key value. The idea of these inconsistent nodes are derived from a relaxed heap (44). In the relaxed heap, these nodes are called *bad children.* In this thesis, the terms *inconsistent node*, *active node* and *bad node* are used interchangebly to represent the same meaning.

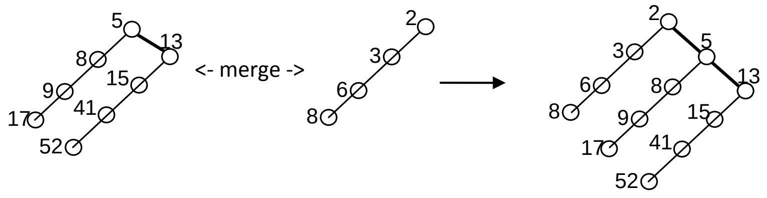
A collection of quaternary trees forms a quaternary heap. The quaternary trees are known by their degrees. The *degree* of a quaternary tree is given by the degree of the root node. Generally, $\mathbf{T}(i)$ is said to be a tree of degree i, which is the degree of the root. The degree of a node can be obtained by calculating the total number of trunks that connect to the node.

A quaternary heap with an underlying polynomial of trees $P = \mathbf{2T}(2) + \mathbf{3T}(1) + \mathbf{3T}(0)$ is shown in Figure 3.4. Trees within the quaternary heap are linked to each other by their roots. A pointer H is used to point to the lowest degree tree in the heap. Inconsistent nodes are indicated by black circles.

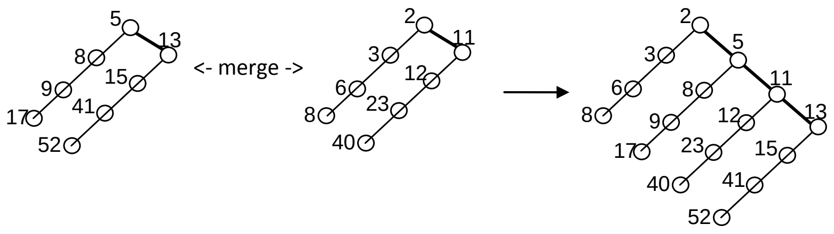
The rightmost tree is the lowest degree tree. The degree of the tree increases from right to left trees in the heap. The quaternary tree $\mathbf{T}(i)$ contains a maximum of $\mathbf{3T}(i-1)$ trees. If the quaternary tree of $\mathbf{1T}(i)$ is merged to $\mathbf{3T}(i)$, the result will be a new $\mathbf{1T}(i+1)$. This means that the degree of the tree is increased by one, thus



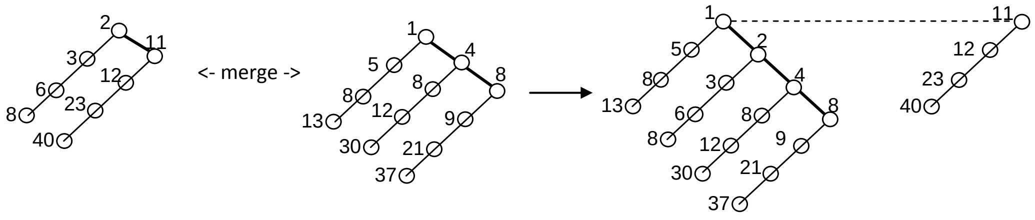
(a) The merge of **1T**(1) tree with another **1T**(1) tree



(b) The merge of **2T**(1) tree with **1T**(1) tree



(c) The merge of **2T**(1) tree with another **2T**(1) tree



(d) The merge of **2T**(1) tree with **3T**(1) tree

Figure 3.3: Merging process involving different types of trees

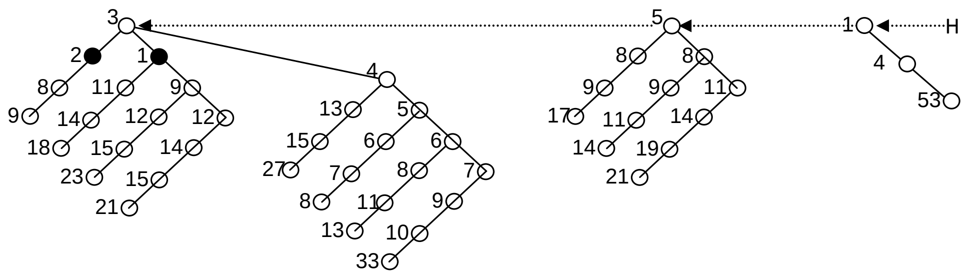


Figure 3.4: An example of a quaternary heap

Lemma 1 *For the quaternary tree **1T**(i), there are 4^i nodes.*

Proof The proof is by induction on i. The basic of the quaternary tree is $\mathbf{1T}(0)$. A quaternary tree $\mathbf{T}(i)$ consists of a maximum of $\mathbf{4T}(i-1)$. Therefore $\mathbf{1T}(i)$ has $4^{i-1}+4^{i-1}+4^{i-1}+4^{i-1} = 4^i$ nodes.

From lemma 1, the number of nodes in the quaternary heap in Figure 3.4 can be calculated as below:

$$\begin{aligned}|P| &= \mathbf{2T}(2) + \mathbf{3T}(1) + \mathbf{3T}(0)\\ &= \mathbf{2} \times \mathbf{4}^2 + \mathbf{3} \times \mathbf{4}^1 + 3 \times \mathbf{4}^0\\ &= 47\end{aligned}$$

A basic node structure in the quaternary heap is shown in Figure 3.5. The structure type used for nodes is very similar to that used for other heaps (3), (4) and (2). The main difference is that below_ partner and above_ partner pointers are used which points to nodes partners on the same trunk. A description of each node's attributes is presented in Table 3.1.

3.2.1 Quaternary heap operations

Common operations supported by the quaternary heap are described. These operations are *insert, delete-min and decrease-key.*

insert(H, x)**:** inserts element x into the heap pointed by the H pointer.

delete-min(H)**:** removes and returns the minimum key value from the heap.

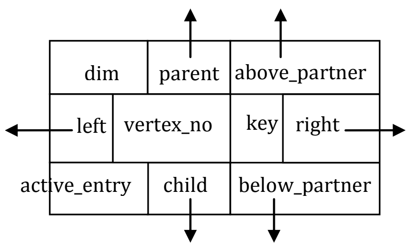


Figure 3.5: A basic node structure in the quaternary heap

Attributes	**Descriptions**
parent	**a** pointer to point to the parent of the node
child	**a** pointer to point to the child of the node.
vertex_no	**the** number of the graph vertex that the node corresponds to
key	**the** key value of the node
below_ partner	**a** pointer to point to the below partner of the node. (Below partner is a node that is located below the node on the trunk)
above_partner	**a** pointer to point to the above partner of the node. (Above partner is a node that is located above the node on the trunk)
dim	**a** dimension of the node is equal to the degree of the node
active_entry	**an** indicator to check node's consistency (If the node is not active this field will be NULL)
left	**a** pointer to point to the left sibling of the same parent
right	**a** pointer to point to the right sibling of the same parent

Table 3.1: The descriptions of each attribute of a node in the quaternary heap

decrease-key(H, x, k): replaces the key value of node x with k value. Here the value of k is always lower than or equal to the current key value of x.

The details of each operation are described in the followings.

3.2.1.1 Insert operation

Insert process is a process to insert a new key into a heap. It may also be defined as a process to merge a new tree of type $\mathbf{T}(0)$ into the heap. The particular process depends on whether there are any existing trees of the same type in the heap. There are four cases that need to be considered:

Case 1 There is no existing tree of type $\mathbf{T}(i)$, then simply insert the new tree into the $\mathbf{T}(i)$ position on the heap at the root level.

Case 2 There is an existing tree of type $\mathbf{T}(i)$ with coefficient value, $\mathbf{a}_i = 1$, that is $\mathbf{1T}(i)$. The insert process will create a new $\mathbf{2T}(i)$. Only one comparison is needed to compare the root values.

Case 3 There are two existing trees of type $\mathbf{T}(i)$ with coefficient value, $\mathbf{a}_i = 2$, that is $\mathbf{2T}(i)$. The insert process will create a new $\mathbf{3T}(i)$. At most two comparisons are needed.

Case 4 There are three existing trees of type $\mathbf{T}(i)$ with coefficient value, $\mathbf{a}_i = 3$, that is $\mathbf{3T}(i)$. The insert process is supposed will create a new $\mathbf{4T}(i)$. However, the coefficient value reaches the limit (that is 4). This new tree structure is called a carry tree of $\mathbf{T}(i+1)$ with coefficient value, $\mathbf{a}_{i+1} = 1$. Thus, the new tree is $\mathbf{1T}(i+1)$. This carry tree is forced to continue the insertion process with the tree of type $\mathbf{T}(i+1)$ at the root level.

Generally, to insert only a single node to the heap, the new node will be added to the rightmost tree, on the $\mathbf{T}(0)$ position. However, contrary to the trinomial heap, an insertion cache is introduced in this quaternary heap. For this purposes, an adaptive cache on the incoming stream of nodes is created to catch up to four consecutive nodes. Each new node will be compared with the largest node held by the cache. Let the current node in the adaptive cache be called a cached node. The next incoming node is a node which is ready to be inserted after the current node. If the key value of the next incoming node is greater than the key value of the cached node, one comparison is needed and the new node will be placed below the cached node on a trunk. The adaptive cache now has two nodes in it.

In the insertion cache, whenever a new node is inserted, the key value of the new node will always be compared with the key value of the lowest node in the adaptive cache. Thus, only one key comparison is needed. If there are many sequences of monotone-increasing key values, there are likely many completed trunks created in the cache. The trunk will then be flushed from the adaptive cache and be merged to the tree $\mathbf{T}(1)$ heap position at the root level. Otherwise, if it resulted in an incomplete trunk, this trunk will be merged to the $\mathbf{T}(0)$ heap position. That is, two entrances to the heap, $\mathbf{T}(0)$ and $\mathbf{T}(1)$ are provided. The cache concept technique can reduce the number of node-to-node key comparisons when inserting the new node into its correct tree position. Figure 3.6 shows some intermediate stage during the insertion operation. In the figure, there is a sequence of incoming stream of nodes that consists of nodes 5, 12, 27, 35, 67, 80 and 6.

The insertion process in the cache takes one comparison. It is expected that 4-node trunks can absorb the effect of partially sorted sequences better than 3-node trunks. In other words,

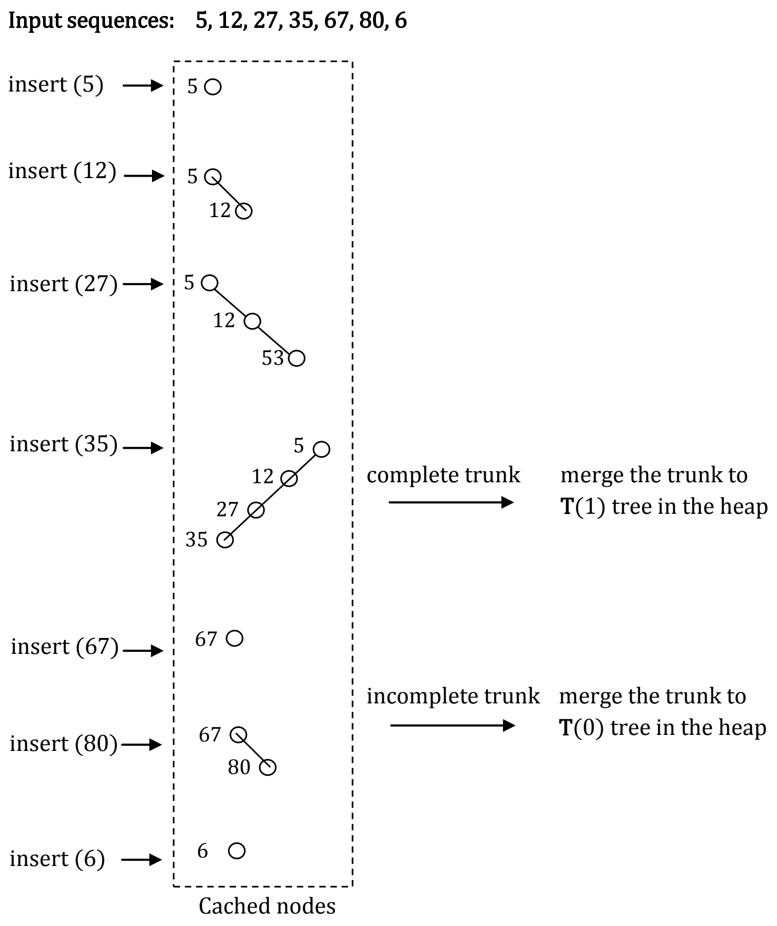


Figure 3.6: Two entrances introduced in the quaternary heap

the quaternary heap can behave adaptively for partially sorted inputs. For other insertions described in the four cases above, different running time is obtained. For cases 1-3, the running time is in $O(1)$ time. However, when a carry tree is created (case 4), the carry key propagates to merge with the higher degree tree. Therefore the insert process is in the worst case in $O(\log n)$ time.

3.2.1.2 Delete-min operation

Delete-min is a process that performs the following three steps: finds the minimum node in the heap, removes the minimum node from the heap, and re-arranges the heap accordingly after the minimum node has been removed. In the quaternary heap, to find the minimum node, three locations should be found. First, the minimum node in the heap can be found by searching the root nodes of all trees in the heap that pointed to by a pointer H. This process takes $O(\log n)$ time. Then, the minimum node should also be searched from the active node list. This also takes $O(\log n)$ time. Lastly, the root node in the cache memory is sought that takes $O(1)$ time. These three minimum nodes from different locations will be compared, and the node with the lowest key value will be chosen as the minimum node that will be removed from the heap.

Once the minimum key is found, it must be deleted from the heap. If the minimum node is found from the cache area or from $\mathbf{T}(0)$, $O(1)$ time is taken to remove the node. No arrangement is made to the heap. The lower node located on the same trunk with the minimum node will be chosen as the new root of the tree, if it exists.

If the minimum node is obtained from the root node, the tree will be broken apart into smaller sub-trees. Let v be a root node of $a_i\mathbf{T}(i)$, where $1 < a_i < 4$ in the quaternary heap. When v is removed, sub-trees, $b_i\mathbf{T}(i)$,$b_i\mathbf{T}(i-1)$,…, $b_i\mathbf{T}(0)$ are obtained, resulted in the removing of v from the heap. The number of these sub-trees depends on the value of a_i of the tree $a_i\mathbf{T}(i)$ rooted at v. However, if $a_i = 1$, then only trees of $b_i\mathbf{T}(i-1)$,…, $b_i\mathbf{T}(0)$ are obtained. In other words, there no sub-tree of $deg(v)$ exists anymore. Figure 3.7 shows an example of a tree that breaks into three separated sub-trees when the root node is removed. Obviously, to merge these sub-trees back to the heap at the root level takes $O(\log n)$ time.

3.2.1.3 Decrease-key operation

To discuss decrease-key operations, two internal operations should be explored. For a trinomial heap, these operations are called reordering and rearrangement. Active nodes are first created due to inconsistency at the beginning, but can become consistent when the key of the head node decreases.

Rearrangement

The arrangement of nodes in a trunk is another technique for reducing the number of active nodes in the heap. Usually this process comes before the reordering process. The *rearrangement* process will rearrange the position of the same dimension active nodes that are located on two different trunks. During the rearrangement process, two or three active nodes of same dimensions are placed on the same trunk and later the reordering process is called. This process is also similar to the rearrangement operation in a trinomial heap. See an example of the rearrangement process, see Figure 3.8.

Steps in the rearrangement are described as follows.

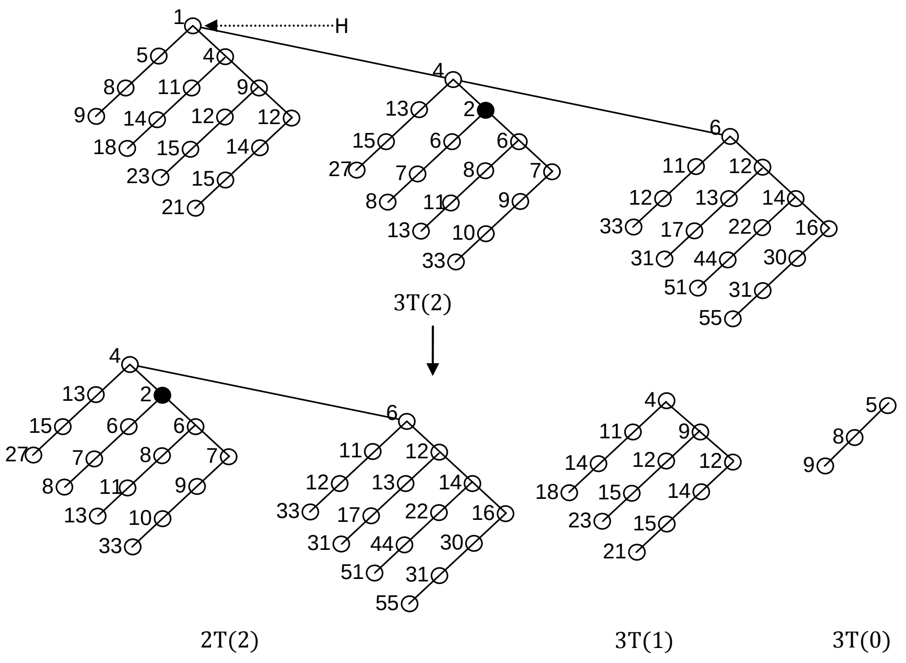


Figure 3.7: Sub-tress, $\mathbf{2T}(i)$, $\mathbf{3T}(i-1)$ and $\mathbf{3T}(0)$obtained when the root node of $\mathbf{3T}(i)$ is removed

1 Identify active nodes, its below partner, above partner, and parent. Mark them with v_1, v_2, v_3 and P_v.

2 If there are two or three inactive nodes on the same trunk, the active nodes have to be made inactive. If the key of v_2 or v_3 are less than the parent key, a promotion or reordering operation is called.

3 The second trunk is checked for other inactive nodes. Here, the trunk is labeled with w_1, w_2, w_3 and P_w.

4 If there are two or three active nodes on the same trunk, the second and the third node are made inactive and perform promotion if necessary.

5 Arrange v_1, v_2, v_3 and w_1, w_2 and w_3 accordingly.

Reordering

Reordering is the process of reordering node position on the same trunk. It is done to reduce the number of active nodes in the tree. Let u, v and w be active nodes on a trunk of dimension i, placed according to the ascending order of key values, $key(u) \leq key(v) \leq key(w)$.

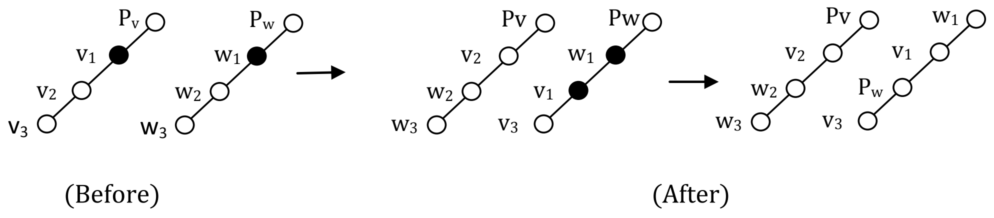


Figure 3.8: Rearrangement process in the quaternary heap

The head node of this trunk is P_u and this node is in dimension $i + 1$. During the ordering process, $key(w)$ will be compared with $key(P_u)$. If $key(P_u) < key(w)$, node w will be made inactive, and nothing else. In this case, one active node is reduced. In a case where only the nodes u and v be active nodes, key value of node v will be compared with $key(P_u)$. Node v will also be made an inactive node if $key(P_u) < key(v)$. Otherwise, P_u is moved to the appropriate position on the trunk, and its dimension is decreased to i. Node u will replace the old position of P_u, and its dimension now becomes, $dim(u) = i + 1$. With a reordering operation, the number of inactive nodes can be reduced. However, replacing P_u with u as the new head node may cause another decrease-key operation at dimension $i + 1$. Figure 3.9 demonstrates the reordering process in a quaternary heap.

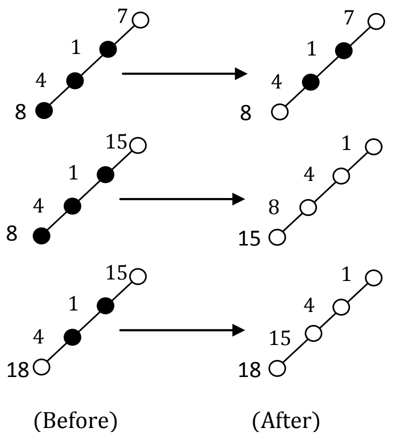


Figure 3.9: Different cases of reordering process in the quaternary heap

The *decrease-key* process in the quaternary heap deploys the same decrease-key technique such in the trinomial heap. The objective of this operation is to decrease the key value of the node to a new key value which is lower or equal to the current key value. Heap violation might occur as the result of this operation. In the quaternary heap, once the key value of a node is less than its parent node, the node is said to be an inconsistent or an active node. The quaternary heap permits certain nodes to be inconsistent nodes. Two or three inconsistent nodes are allowed to be on the same trunk or even of the same dimensions. However, the number of inconsistent nodes is limited and it is always kept under control by performing two operations: reordering and rearrangement as explained before.

There are a few cases of decrease-key operation. It depends on the position of the decreased node, v on a trunk.

Case 1 If v is a root node, rearrangement is not necessary as the key value of root is always smaller than other nodes.

Case 2 This case involves active nodes. If v is the first child and it was an active node, rearrangement is not necessary as the number of active nodes is maintained. If v is the middle or the last node on a trunk and its key has become less than the upper node(s) on the trunk, which were active nodes, v is swapped with the upper node(s)to maintain the correct ordering of the heap. Then, v is made an active node.

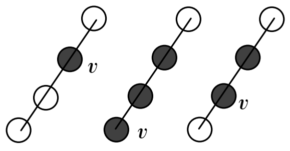


Figure 3.10: Different v's positions

Case 3 In this case, v is not an active node and it is located on the main trunk. If the key value of v is smaller, it has to be swapped with other key values to maintain the correct heap ordering. After it has been swapped, v might become the root node, therefore the tree of that particular dimension is made rooted at v.

Case 4 This is a complete trunk where there is a parent node. Node v is located somewhere on the trunk and it can be the first node, the middle node or the last one. If v is the middle or the last node on the trunk, v is swapped with the upper node(s) on the trunk

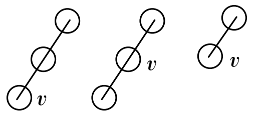


Figure 3.11: Different v's positions on the main trunk

to maintain the correct ordering. In a case where v is the first child or becomes the first child after the key value is decreased, v is made an active node. The number of active nodes might reach the tolerance level after v becomes active. Thus, rearrangement might need to be performed to control the number of active nodes in the heap.

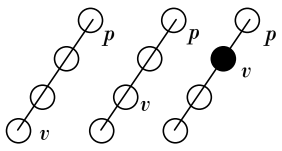


Figure 3.12: Different v's positions on a complete trunk

The number of active nodes in the quaternary heap is maintained within tolerance. To do that, a counter is used to count the total number of active nodes. If the total number of active nodes exceed the allowed number, rearrangement or/and reordering operations will be performed. Using the pigeon hole principle, it is known that if the number of inactive nodes reaches its limitation, then, there must be at least two active nodes of the same dimensions existed. Thus, this will cause no chain effect. For the quarternary heap, decrease-key operation is done in $O(1)$ worst case time.

In the next section, another new heap is explained. If the quaternary heap always maintains the dense structure, the next invented heap tries to maintain the thinnest structure possible.

3.3 A Dimensional Heap

A *dimensional heap* is a collection of trees that are based on binary linking and satisfy the minimum heap property. This implies that an element with the lowest key value is always at the root level. Just like the existing 2-3 heap, the dimensional heap is constructed by binary linking of trees repeatedly, that is, repeating the process of making the product of linear tree and a tree of lower dimension.

Each tree in the heap consists of nodes and branches. Every two nodes are connected to each other by a single line that is called a branch. Note that some common heap data structure terminology applied here are equivalent to the existing terms defined in Table 3.1, unless it is stated in another meaning. A node is said to be in dimension 0 if the node is a leaf node. This means the node has no child node underneath. When any node becomes a parent node, the dimension of the node is changed to a higher one. In the dimensional heap, a parent node in dimension i can have a maximum of 2 children nodes in each of dimensions $i-1, i-2, ..., 0$. Each child of a parent is connected to each other by left and right pointers. These children are called siblings. The other existing heaps such as 2-3 heap (4) and trinomial heap(2) share almost the same structure as the dimensional heap. Compared to other heaps, a new indicator is introduced in the dimensional heap that is called *thickness*. This indicator is used to check whether the node has any sibling that is in the same dimension. If the same dimension siblings are found, the thickness of the node is set to be true and false otherwise.

A basic structure of a node in the heap is shown in Figure 3.13.

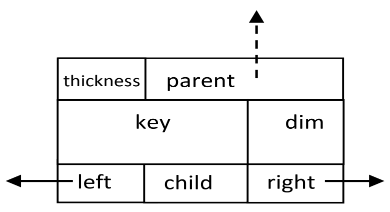


Figure 3.13: A basic node structure in the dimensional heap

A dimensional heap of dimension $n-1$ is given by

$$\mathbf{a}_{n-1}\mathbf{T}(n-1) + ... + \mathbf{a}_1\mathbf{T}(1) + \mathbf{T}(0) \tag{3.3}$$

From 3.3 $\mathbf{T}(i)$ is a tree of degree i and $\mathbf{a}_i$ is a linear tree co-efficient. A symbol $\mathbf{T}$ used here represents a tree. Each $\mathbf{a}_i$ is either 0 or 1. If $\mathbf{a}_i = 0$, that means no existence of tree $T(i)$. The tree of $\mathbf{T}(i)$ exists if only $\mathbf{a}_i = 1$. If there are two children of dimension i, the dimension is said to be thick and they are called thick siblings.

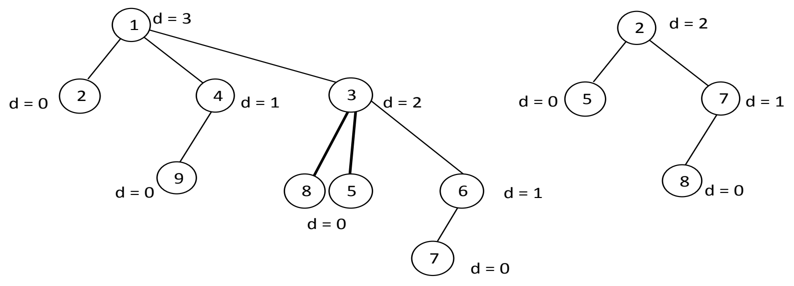


Figure 3.14: An example of a dimensional heap (d indicates *dimension*)

Figure 3.14 shows a heap that consists of two trees, $\mathbf{T}(3)$ and $\mathbf{T}(2)$. The nodes are identified by their key values for simplicity. The root nodes are nodes with key values (1) and (2). There are two *thick* edges from a node with key value (3), that is $(3, 8)$ and $(3, 5)$. A tree $\mathbf{T}(3)$ has 3 children of dimensions 2,1 and 0. The lowest dimension child, that is a child node in dimension 0, is always located at the left most location of the children, while the highest dimension child is located at the right most of the tree (Figure 3.14 uses d to represent a dimension). The thick lines used in the Figure is to indicate the thick edges. The tree is called a complete tree if it has two children in each dimension as shown in Figure 3.15. All edges in the complete tree are only thick edges. For the sake of clarity, the tree structure of $\mathbf{T}(3)$ in Figure 3.14 is provided in Figure 3.16.

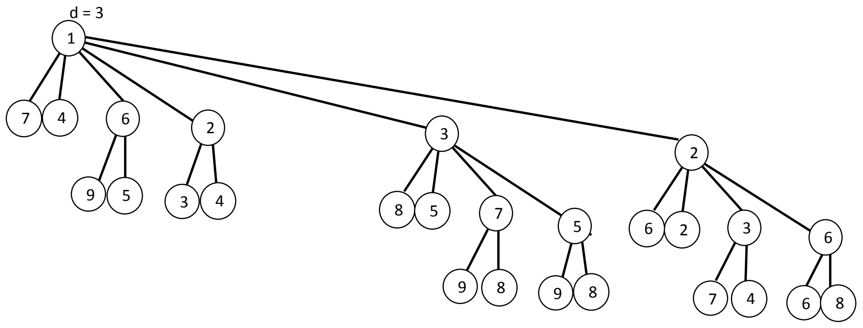


Figure 3.15: An example of a complete dimensional heap

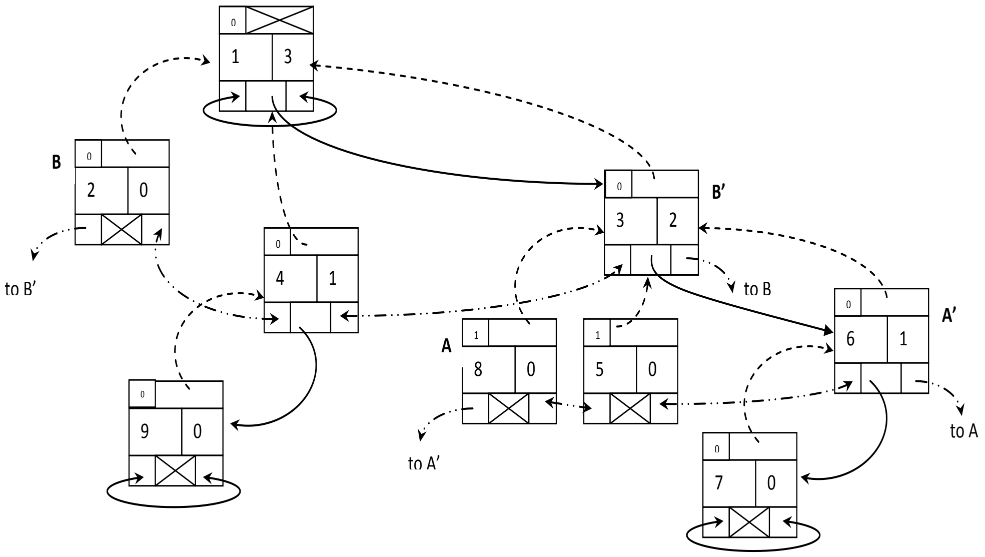


Figure 3.16: Internal representation of node connectivity in $\mathbf{T}(3)$ in Figure 3.14

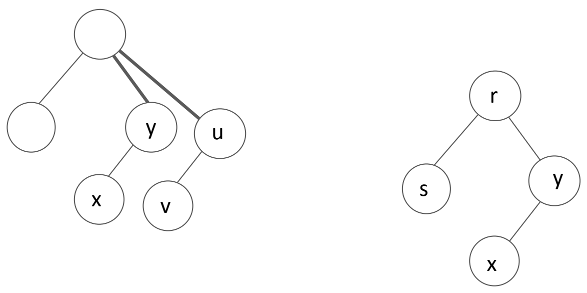
The next section discusses a workspace, tree potential, and amortized cost concepts that are essential before dimensional heaps operations are described.

3.4 A Workspace

A *workspace* of node x is a term used to define four neighboring nodes of x. The workspace of node x with dimension of $x, dim(x) = i$ consists of two nodes of $dim(i)$ and two other nodes in higher dimensions. These higher dimension nodes must be one of $dim(i + 1)$ that defines as a parent of x and the other one is a sibling of the parent or a parent of the parent.

To find the workspace of x, first select the node itself to be the primary node in the workspace. Second, traverse to the parent's node and choose the parent as the next node in the workspace. Third, if the parent labeled as y node has a thick sibling, traverse to the parent's thick sibling and select the node as the third one. Let that node be called node u. Finally, choose u's child, i.e. v, as the fourth node in the workspace. This workspace can be called the right workspace of x.

If the right workspace does not exist, as there is no node u in the tree, the left workspace is looked for. To find the left workspace, traverse to the parent's parent labeled with r. Choose node r as the third node in the workspace. The last node to be chosen is the parent's left sibling, s that has a lower dimension than the parent but apparently same dimension with x. Figure 3.17 shows different workspaces of node x.



(a) An example of the right workspace of x (b) An example of the left workspace of x

Figure 3.17: Workspace definition of node x

There is a case where the workspace can not be reached. In this case, the node itself might be the root node of the tree or the parent of the node has no other sibling. The workspace is not defined in these nodes as they are located at the highest dimension of the tree. Any operation occurs at these nodes will effect the tree structure, whether the tree grows or shrinks.

That means, the tree may no longer remain within standard arrangement for its respective dimension.

Every node in the dimensional heap can use their workspace nodes to assist them in performing an expensive heap operation, as such when the decrease-key function is executed.

3.5 Tree Potential

A potential, Φ, of the dimensional tree is calculated based upon summing the total number of edges in the tree. When there are two nodes on a trunk, the trunk is said to have one potential, $\Phi = 1$. In a tree, some nodes have a thick sibling, which means that these nodes share the same parent and have the same dimension of nodes. These thick siblings are connected to the parent's node by thick edges. For the each thick edge, the potential is defined as $\Phi = 1$ as well. These thick siblings are in the same dimensions. Let e_t and e_k represent the total number of thin edges and thick edges. If a tree, $\mathbf{T}(i)$ has the thin and thick edges, the total potential in given by:

$$\Phi_{T(n)} = e_t + e_k$$

The total number of potential in the heap in Figure 3.14 that has two trees, $\mathbf{T}(2)$ and $\mathbf{T}(3)$ is calculated as below:

$$\begin{aligned}&\mathbf{T}(2) : e_t = 3,\ \text{thus, } \Phi_{\mathbf{T}(2)} = 3\\&\mathbf{T}(3) : e_t = 6, e_k = 2,\ \text{thus, } \Phi_{\mathbf{T}(3)} = 6 + 2 = 8\end{aligned}$$

To count the total number of potential in the heap is to sum the total potential for all trees in the heap. Therefore, the total number of potential in the heap is

$$\sum \Phi_n = 3 + 8 = 11$$

The potential concept will be used in the next section.

3.6 Amortized Cost Analysis

Amortized cost is used to analyze the time taken per operation. The idea of using this analysis is to get the average over the sequence of operations. When running a large program, many operations are involved. Some operations are very expensive to run and some other operations are relatively cheap. However, the number of frequency in running the both operations are different; some cheap operations are used more and occur more frequently compared to expensive ones. With the amortized cost concept, it is somehow guaranteed that the time taken to run a program is efficient.

In a tree or a heap data structure, two main elements are used to measure the amortized cost. The first element is the difference of the potential of the tree before and after the measured operation is called and the second element is the number of key comparisons used by the operation. The potential of a tree is defined as the sums of edges in the tree, while key comparisons are calculated when the operation compares two or more key values of nodes.

Denote Φ_i as the potential of a tree after the i-th heap operation. An amortized cost of the i-th after the i-th operation in a tree is defined $a_i = t_i - (\Phi_i - \Phi_{i-1})$, where a_i is the amortized cost of the operations, t_i is the total number of comparisons calculated for performing the operations, Φ_{i-1} is the potential before the operations are performed and Φ_i is the potential after the operations have been accomplished. The sum of the amortized costs of heap operations gives the overall amortized cost A, which is

$$A = \sum_i a_i$$

Meanwhile, the number of key comparisons gives overall actual cost for the heap operations. Thus, the total costs of heap operations is given by:

$$T = \sum_i t_i$$

The total amortized costs over N heap operations gives:

$$\begin{aligned}
A &= a_1 + a_2 + \ldots + a_N \\
&= (t_1 - (\Phi_1 - \Phi_0)) + t_2 - (\Phi_2 - \Phi_1) + \ldots + (t_N - (\Phi_N - \Phi_{N-1}) \\
&= t_1 + t_2 + \ldots + t_N + (\Phi_N - \Phi_0) + ((\Phi_1 - \Phi_1) + (\Phi_2 - \Phi_2) + \ldots + (\Phi_{N-1} - \Phi_{N-1})) \\
&= t_1 + t_2 + \ldots + t_N + (\Phi_N - \Phi_0) \\
&= T + (\Phi_N - \Phi_0)
\end{aligned}$$

where T is the total number of key comparisons or the total of actual cost, Φ_0 is the heap's initial potential and Φ_N is the potential of the last state. At the starting state, the potential Φ_0 is zero and end state is the same, then $\Phi_N - \Phi_0 = 0$. Therefore, the total amortized cost is reduced to:

$$A = T$$

that is the total amortized cost of heap operations is equal to the total of actual cost.

From this analysis, there are three possible outcomes to be achieved, in terms of whether each a_i is positive, 0 or negative. If positive value is achieved, it means that a cost was incurred during the operation. Negative result on the other hand means a profit was gained during the operation. If a zero result is obtained, the cost of the particular operation is essentially free. An example of a simple amortized cost calculation is described below.

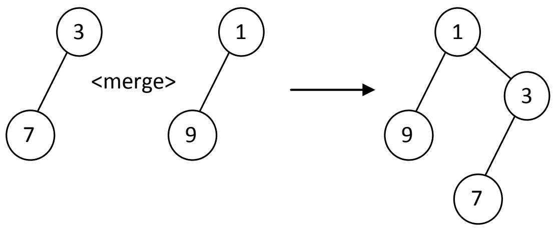


Figure 3.18: Merging of two trees, $\mathbf{T}(1) + \mathbf{T}(1)$, resulted in a new $\mathbf{T}(2)$

Figure3.18 shows the merge of two trees of $\mathbf{T}(1)$. The potential of each tree before merging is 1, thus, for the two trees, $\Phi = 1 + 1 = 2$. To merge the tree, one comparison is needed to compare the root nodes. After the merge operation, a new $\mathbf{T}(2)$ is created. This new tree has a potential, $\Phi = 1 + 1 + 1 = 3$. The amortized cost for a single m merge operation is:

$$\begin{aligned} a_m &= t_m - (\Phi_m - \Phi_{m-1}) \\ &= 1 - (3 - 2) = 0 \end{aligned}$$

The motivation for amortized analysis is that implementing an expensive and tricky operation has a lot of cheap operations before it. Using this concept, the worst case analysis of each operation can be said to not estimate the overall performance.

3.7 The Dimensional Heap Operations

In this section, several basic operations in the dimensional heap are given.

Merge: Compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.

Insert: Create a new node in dimension 0 and place to the heap in tree 0.

Delete-min: Find the minimum key value at root level in each tree in the heap and remove the node that has the minimum key value from the heap.

Decrease-key: Decrease the key value of the required node and do some tree arrangement of the heap.

3.7.1 Merge Operation

Given a dimensional heap of dimension i as $\mathbf{a}_{i-1}\mathbf{T}(i-1)+...+\mathbf{a}_1\mathbf{T}(1)+\mathbf{T}(0)$. To expand the trees in the heap means to add a new node to the heap. If the lowest dimension tree of the heap already has a node on it, then the new node must be merged with the existing one. This is how the merge operation comes to play. Generally, to link a minimum of two nodes together requires a merge operation. There are two cases which arise when the merge function is called.

case $a_i = 0$: The new node or tree is simply added in the correct $\mathbf{T}$ position.

case $a_i = 1$: A carry key of $\mathbf{T}(i)$ is made with one key comparison, and increases the potential by one.

Two trees in same dimensions can be merged by comparing the root nodes' key values. Given two trees that are called A and B, the idea is to combine these A and B trees together. The merge process is done as follows. First, both of the trees must be in same dimensions. Next, the key values of the root nodes in tree A and tree B are compared. If the key value of the root node of tree A is less than the key value of the root node of the second tree, the root node of A becomes the new root node of the new tree, or the root node of B otherwise. Note that when the roots are merged, the trees underneath also move accordingly. See Figure 3.19.

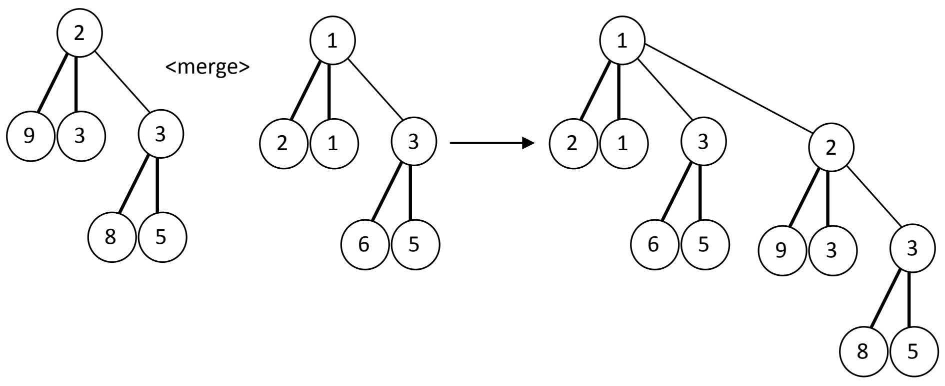


Figure 3.19: An example of a merge process

The result of merging two trees is a new higher dimension tree. For example, as shown in Figure 3.19, when trees of dimension 2 are merged, a new tree $\mathbf{T}(3)$ is created. With this technique, the previous effort used to create the existing tree branches of each $\mathbf{T}(2)$ is not wasted.

The amortized cost for this operation is always free, which is 0. This is because, when two trees are merged, one comparison is needed to choose the new root node and the potential value is one as one new branch is created after the merge. Therefore, the amortized cost for one merge operation is $a_1 = t_1 - (\Phi_1 - \Phi_0) = 1 - 1 = 0$.

3.7.2 Insert Operation

Inserting or adding a new node to a heap is the most basic operation and must be performed at the early stage after initializing the heap. It has to be done at least once, before other operations such as merge, delete-min or decrease-key operations is called. To insert a new node to the heap means to add the node to the lowest dimension tree, that is $\mathbf{T}(0)$.

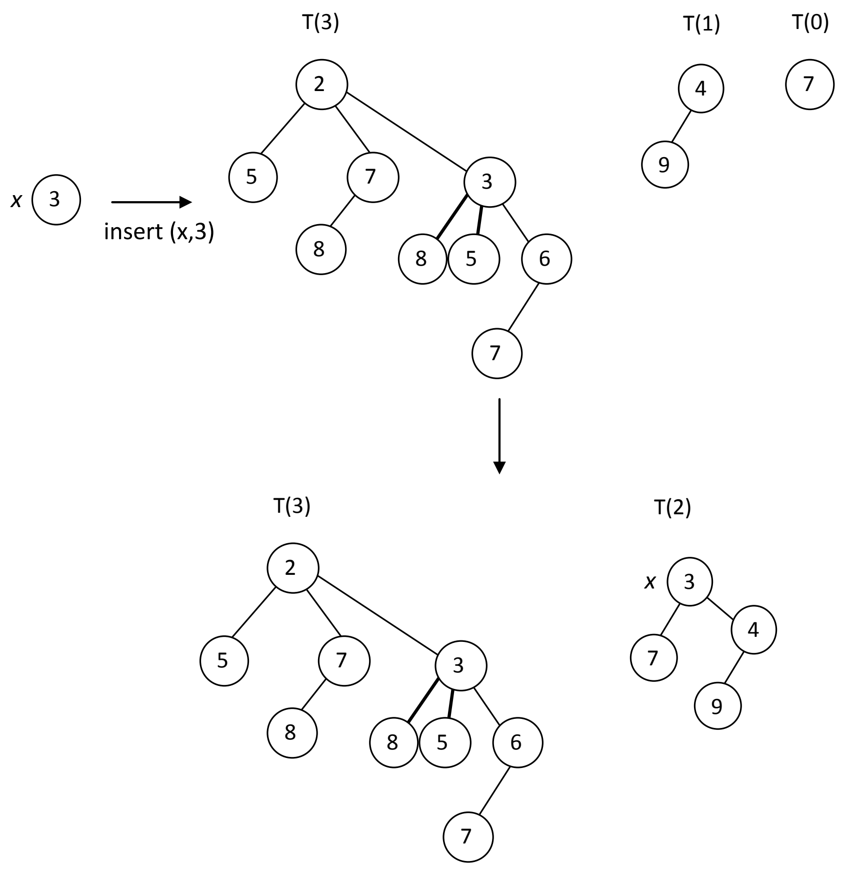


Figure 3.20: The process of inserting node x with $key(x) = 3$ to the existing heap

Let x be the new node to be added to the heap. If $\mathbf{T}(0)$ is empty, insert x at $\mathbf{T}(0)$ and x becomes the only node in $\mathbf{T}(0)$. If there is the existing $\mathbf{T}(0)$ in the heap, $key(x)$ is compared with the key value of root node of $\mathbf{T}(0)$. If $key(x)$ is less than or equal to the key of the root node, make a carry tree to $\mathbf{T}(1)$ with x as the new root node. This may propagate to the higher $\mathbf{T}(i)$ if there are existing trees of $\mathbf{T}(1), \ldots, \mathbf{T}(i-1)$ in the heap. Figure 3.20 explains the steps taken when node x with $key(x) = 3$ is added to the heap. In this example, the current heap has trees of $\mathbf{T}(0), \mathbf{T}(1)$ and $\mathbf{T}(3)$.

The insert operation works as follows. Firstly, $key(x)$ is compared with (7) at $\mathbf{T}(0)$. Secondly, they are merged to create a new $\mathbf{T}(1)$ with x as the root node. Note that $key(x)$ is less than or equal to the previous $\mathbf{T}(0)$ root key value. When the new $\mathbf{T}(1)$ is created, $\mathbf{T}(0)$ is released to be empty as there is no more node in $\mathbf{T}(0)$. Thirdly, the new $\mathbf{T}(1)$ will be merged with the existing $\mathbf{T}(1)$ in the heap after comparing $key(x)$ and (4). As $key(x) \leq (4)$, x is chosen to be the root node of the new $\mathbf{T}(2)$ as the result of merging trees of $\mathbf{T}(1)$. Finally, the propagation ends as there is no more tree of $\mathbf{T}(2)$.

The amortized cost to insert a new node in the heap is always free. If there is an existing node in $\mathbf{T}(0)$, one comparison is needed and one potential is gained, making the amortized cost zero. The process is also free if there is no node in $\mathbf{T}(0)$ as no comparison neither potential are used or gained with this process. In the worst case, insert operation is done in $O(\log n)$ time as the result of propagating $\mathbf{T}(i)$ to $\mathbf{T}(i+1)$.

3.7.3 Delete-min Operation

The delete-min operations is described in details. The use of delete-min operation is to return the minimum key from the heap. To do this, delete-min must search for the smallest key value at the root level in each tree. When the node with the minimum key is found, the operation will break apart all children of the minimum node and merge them to the appropriate trees.

Let the root of tree $\mathbf{T}(i)$ have the minimum key. Children of tree $\mathbf{T}(i)$ are trees of $\mathbf{T}(i-1), \mathbf{T}(\ldots), \mathbf{T}(0)$. These children trees will be disconnected from the minimum node that is their parents' node. These trees will then be merged to the trees of the same dimension in the heap. For example, the tree $\mathbf{T}(i-1)$ will be merged to the existing tree $\mathbf{T}(i-1)$ in the heap. The merge concept is similar to the description of merging operation in 3.7.1.

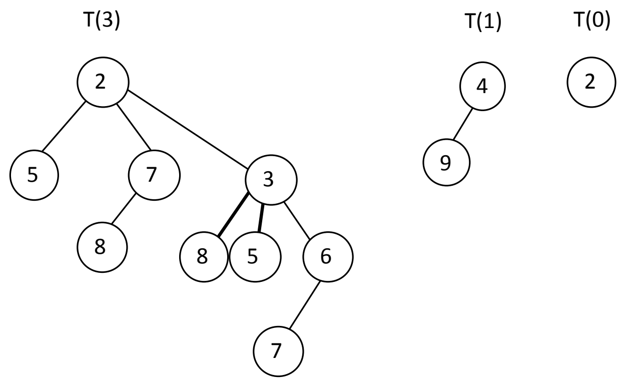


Figure 3.21: The result after performing delete-min on Figure 3.14

For clarity purposes, Figure 3.14 is referred to. Let the delete-min operation be called. The process will first compare the key value of (2) and (1) that are located at the root level. The node with the key value (1) is chosen, thus delete-min process occurs at $\mathbf{T}(3)$. The children nodes, $\mathbf{T}(0)$ with the root key value (2), $\mathbf{T}(1)$ with the root key value (4) and $T(2)$ with the root key value (3) will be cut off from the $\mathbf{T}(3)$. As the heap has none existing $\mathbf{T}(0)$ and $\mathbf{T}(1)$ trees, these broken trees, i.e. $\mathbf{T}(0)$ and $\mathbf{T}(1)$ as a result of cutting off from the minimum node will become the new tree of $\mathbf{T}(0)$ and $\mathbf{T}(1)$ in the heap. However, the child tree of $\mathbf{T}(2)$ will be merged to the existing $\mathbf{T}(2)$ tree in the heap. The result after the delete-min process occurred is demonstrated in Figure 3.21.

The amortized cost for one delete-min operation is $3\log n$ as one $\log n$ comes from the fact that to search for the minimum node requires $\log n$ time. The other $2\log n$ comes from cutting branches under the minimum node.

3.7.4 Decrease-key Operation

The decrease-key operation is used to update the key value of a node in the heap. The updating process reduces the key value to a smaller one. When the key value of a node is decreased, the current structure of a heap might violate as the heap does not follow the minimum heap property.

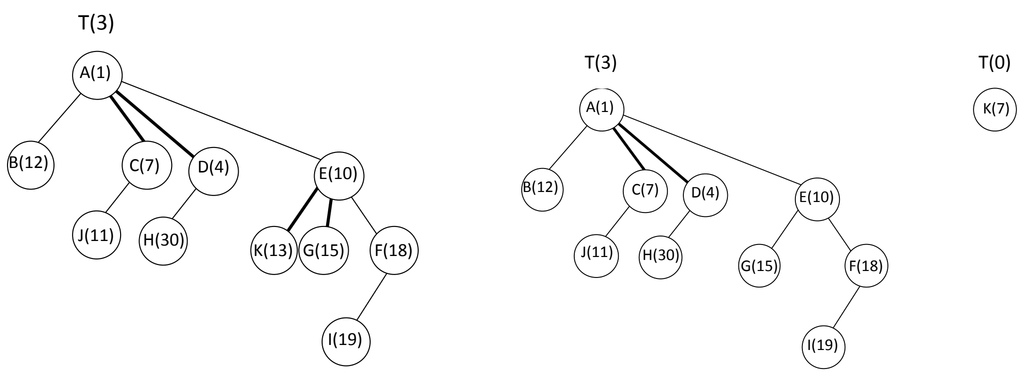
In the dimensional heap, a decrease-key process requires the workspace of a decreased node to be identified beforehand. However, there are two special cases which frequently occur and make the decrease-key of the dimensional heap best implemented, thus not requiring any information about the workspace. The cases are:

case 1 : The decreased node is located at the root level position.

case 2 : The decreased node has a thick sibling

Case 1: When a key of a root node is decreased, nothing is changed, as the structure of the tree remains the same. This is because, the new key value is always lower than the existing key value when the decrease-key process occurred.

Case 2: Let a decrease-key operation is performed on node v. If the parent of v has other child of same dimension as v or in other words, thick edges present, cut tree(v) and move v with its subtree to the root level. The thickness of node v and the sibling node becomes false and only one thin edge remain. To show and example of this case, Figure 3.22 is referred. In this Figure, the node is labeled together with the key value in bracket to make the explanation easier.

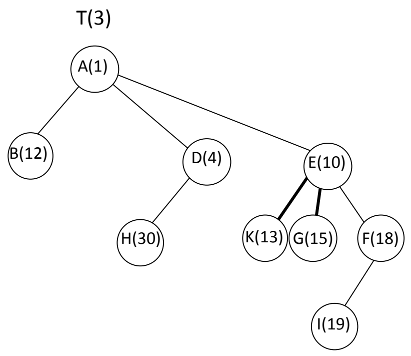


(a) Before $key(K)$ is decreased (b) The result after $key(K)$ was decreased

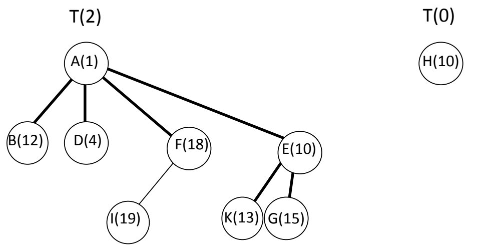
Figure 3.22: Performing a decrease-key operation on node K with the new $key(K)$ value

For an explanation of the **case 2**, let a decrease-key is performed on node K with a new key value, $key(K) = 7$. Make the $key(K) = 7$ replacing $key(K) = 13$. Then, the edge (E, K) is cut off from E. The merge process is called to merge K into an appropriate position at the tree level. The edge of (E,G) becomes a thin one as only one edge of dimension 0 left in the tree.

Other cases: to perform a decrease-key, a workspace nodes are needed. Let the parent u of v be on the i-th dimension of its parent. The decrease-key operation is called to reduce $key(v)$. If u has one child, i.e. v, move u to the lower dimension if the lower dimension is thin. If node u itself be the only child of dimension i, as a result of relocation of u, the parent of u loses dimension i. The effect of this will propagate to higher dimensions. If the current tree's dimension was n and the adjustment makes it $n-1$, the resulting tree will be inserted to $\mathbf{T}(n-1)$. However, if the lower dimension of u is already thick, move one child to dimension i, make binary linking and recover the heap property. With one comparison, the heap property can be recovered. Figure 3.23 and 3.24 show the result when $key(H)$ is decreased. In Figure 3.23, a parent of the decreased-node has a thin lower dimension node while in Figure 3.24, the parent of H has thick nodes in the lower dimension. Note that the existing tree is shrunk as the result of the decrease-key operation in Figure 3.23, resulting in a new $\mathbf{T}(2)$ that replacing $\mathbf{T}(3)$.

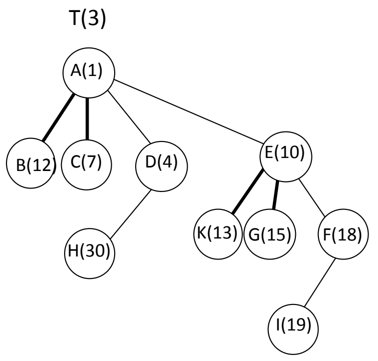


(a) Before $key(H)$ is decreased

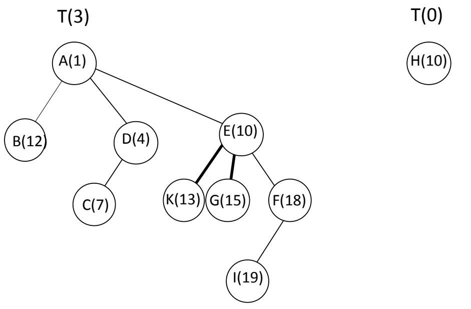


(b) The result after $key(H)$ was decreased

Figure 3.23: Decrease-key operation on node H. The lower dimension sibling of parent's node is thin



(a) Before $key(H)$ is decreased



(b) The result after $key(H)$ was decreased

Figure 3.24: Decrease-key operation on node H. The lower dimension sibling of parent's node is thick

The amortized cost a_i for the i-th operation is defined by $a_i = t_i - (\Phi_i - \Phi_{i-1})$.

Decrease-key of node at the root level : The amortized cost is zero as nothing is changed.

Decrease-key of thick node : To cut the node from the tree will decrease the potential from 2 to 1, $\Phi = 2 - 1 = 1$. As defined in the earlier section, the potential for a pair of thick edges is 2. If one of the thick edges is cut, only one thin edge remains, hence, the potential becomes 1. When the decreased node is merged to the root level of dimension i, if there is no existing $\mathbf{T}(i)$, no key comparison is made. Otherwise, one comparison is needed and the potential is increased by 1. Let us consider decrease-key at node K in Figure 3.22.

$$\begin{aligned} a_i &= t_i - (\Phi_i - \Phi_{i-1}) \\ &= 0 - (9 - 10) \\ &= 1 \end{aligned}$$

Relocate node to the lower dimension node as shown in Figure 3.23 : No comparison is made, thus the amortized cost:

$$\begin{aligned} a_i &= t_i - (\Phi_i - \Phi_{i-1}) \\ &= 0 - (7 - 8) \\ &= 1 \end{aligned}$$

Swapping a thick node with the decreased node as shown in Figure 3.24 : One comparison is required to compare the key value of the parent's node with its thick sibling on the left. The amortized cost is as follows:

$$\begin{aligned} a_i &= t_i - (\Phi_i - \Phi_{i-1}) \\ &= 1 - (8 - 9) \\ &= 2 \end{aligned}$$

The amortized complexity of decrease-key is constant.

3.8 Experimental Results and Analysis

This section presents the results of the experimental comparison of different heaps data structures. Section 3.8.1 contains the number of key comparison results for the quaternary heap. Section 3.8.2 presents the number of key comparisons of the dimensional heap.

The experiment were initially done using an Intel(R) Core(TM) 2 Quad CPU Q8400 @ 2.66Ghz, 3.24 Gb of RAM machine, running Fedora Linux operating system, at the University of Canterbury, New Zealand. All data structure implementations were written in the C programming language. These programs were compiled using the gcc compiler. All of the results from the experiments reported in this chapter were collected on the sane hardware. In the experiments which were carried out, only 10 samples of graph were used.

To see the validity of samples needed to get the results, further experiments were run using an Intel(R) Xeon (R) CPU E5645 @ 2.40Ghz, 4.0 Gb of RAM machine, running on Ubuntu Linux operating system, at Sultan Idris Education University, Malaysia. In this experiment, a binary heap was used as it is a well known heap data structure and widely used in real life applications. Sparse digraphs were used as an input with the average outgoing edges from each graphs were four. The average total number of key comparisons needed to solve Dijkstra's Single Pair Shortest Paths (SPSP) problem was recorded. Standard deviation (SD) and coefficient of variation (CV) were also calculated. The CV is defined as the ratio of the standard deviation to the mean. With the CV, the dispersion of the total key comparisons around the mean of the total number of key comparisons can be measured. For the experiments, the number of samples chosen were 10, 50 and 100. The results can be seen in Tables 3.2.

Even though the number of samples are different, very similar results were obtained. The coefficient of variations, CVs for all the results were less than 1%. As the value of coefficient of variation is low, the results have less variability and high stability. Therefore, it can be suggested that ten samples are enough to run the experiments to compare the performances of different heaps with Dijkstra's algorithm.

3.8.1 The performance of the quaternary heap

Experiments were conducted to see the number of key comparisons between the quaternary and the trinomial heaps. Only the trinomial heap has been chosen to be compared with the quaternary heap because it is very comparable to the quaternary heap. The idea of the quaternary heap is also based on the existing trinomial heap. The way they were implemented is the same. The only difference between the two heaps is trunk size. The maximum number of nodes allowed in each trunk is three in the trinomial heap, whereas, in the quaternary heap, the trunk size is extended to have one more node. That means, the maximum number of nodes in the quaternary heap's trunk is four. The quaternary heap also requires more storage and manipulation of six pointers per node compared to five pointers per node in a trinomial heap.

(a) Sparse digraphs with samples, $s = 10$

Input Size, n	**Min** ($\times 10^3$)	**Max** ($\times 10^3$)	**Mean** ($\times 10^3$)	**SD** ($\times 10^3$)	**CV**(%)
2000	24.07	24.52	24.28	0.11	0.47
4000	52.41	52.83	52.67	0.13	0.25
6000	82.35	82.84	82.56	0.16	0.19
8000	113.22	113.78	113.52	0.18	0.16
10000	144.32	145.12	144.67	0.25	0.17

(b) Sparse digraphs with samples, $s = 50$

Input Size, n	**Min** ($\times 10^3$)	**Max** ($\times 10^3$)	**Mean** ($\times 10^3$)	**SD** ($\times 10^3$)	**CV**(%)
2000	24.06	24.60	24.33	0.11	0.47
4000	52.21	53.10	52.70	0.17	0.33
6000	81.86	83.15	82.60	0.24	0.29
8000	113.03	114.05	113.42	0.24	0.22
10000	144.19	145.65	144.77	0.26	0.18

(c) Sparse digraphs with samples, $s = 100$

Input Size, n	**Min** ($\times 10^3$)	**Max** ($\times 10^3$)	**Mean** ($\times 10^3$)	**SD** ($\times 10^3$)	**CV**(%)
2000	24.01	24.60	24.33	0.12	0.49
4000	52.26	53.11	52.71	0.19	0.36
6000	81.92	83.00	82.53	0.22	0.26
8000	112.80	114.23	113.37	0.27	0.24
10000	144.00	145.31	144.70	0.27	0.18

Table 3.2: The total number of node-to-node key comparisons needed in a Binary heap when solving the SPSP problem using different samples of sparse digraphs

The goal of developing the quaternary data structure is to see the performance of the heap over the existing trinominal heap. In other words, with a dense structure that has more nodes on one trunk, does the new data structure perform better in term of number of key comparisons? In this chapter, a *dense structure* refers to a heap that has more nodes on a trunk, while a *sparse structure* is the opposite of the dense structure. In the quaternary heap, the concept of adaptive cache to keep the nodes temporary in it before they are flushed to the heap structure is used. When a special type of input sequences that are known in advanced are given to the data structure, does this contribute to the good performance of the data structure? In this case, if the input stream is in an ascending order, a complete trunk can be created in the adaptive cache which later will be flushed straight away to $\mathbf{T}(1)$.

While designing the quaternary heap, it was conjectured that the data structure might have a potential to perform better than the trinomial heap as more complete trunks could be created. It was quite curious to see whether better performance would be shown when a dense type of data structure was used.

To answer these question, a well known algorithm to solve the single source shortest path problem, which is Dijkstra's algorithm was used as the main algorithm. Firstly, dense digraphs with different input sizes were employed and the edge costs were randomly generated. The edge cost of the digraphs were then sorted in ascending order beforehand. This was to ensure that the input used were in an ascending order of sequences in advance.

When a large problem size, n was used, trinomial heap always showed less number of key comparison than the quaternary heap, even as a minimum of $n = 100$. Therefore, when running the experiment, the number of n was reduced to the minimum value to see whether was there any chance that the quaternary heap could beat the trinomial heap.

The quaternary heap with a simple insertion process was compared first with the trinomial heap. The results are shown in Table 3.3. The units numbers are node-to-node key comparisons.

Table 3.3 shows that when the total number of vertices, n, is very low, that is $n \leq 30$, the quaternary heap gives a lower number of key comparisons compared to the trinomial heap. However, when n becomes bigger, the trinomial heap becomes superior. In this experiment, the insertion process in the quaternary heap is similar to the trinomial heap.

In the simple insertion process, every new node will be inserted into $\mathbf{T}(0)$. The key of the new node will be compared with the root node first before comparing with other nodes on a trunk in $\mathbf{T}(0)$. If there are three nodes on the trunk, that means, at most 3 key comparisons have to be performed on a trunk at $\mathbf{T}(0)$. A complete trunk can be created as a result of the

Input Size, n	**Trinomial Heap** $(\times 10^3)$	**Quaternary Heap** $(\times 10^3)$
10	359.60	317.20
20	2739.40	2608
30	8369.00	8292.20
40	17191.40	17413.40
50	30310.00	31097.00
60	46691.00	48087.60
70	68498.40	72994.40
80	93571.00	100980.80
90	127390.60	133122.00
100	164913.40	172787.80

Table 3.3: The total number of node-to-node key comparison between the trinomial and the quaternary heaps. The insertion process for each data structure is similar

insertion of a new node. This complete trunk will be merged to the $\mathbf{T}(1)$, and thus, to other higher tree degree. This is the main reason why the number of key comparisons is higher in the quaternary heap than the trinomial heap.

After some modifications were made to the insertion process in the quaternary heap, different results were obtained, as shown in Table 3.4.

Input Size, n	**Trinomial Heap** $(\times 10^3)$	**Quaternary Heap** $(\times 10^3)$
10	359.60	247.20
20	2739.40	2508.40
30	8369.00	7918.00
40	17191.40	16940.20
50	30310.00	29497.80
60	46691.00	45972.20
70	68498.40	68049.80
80	93571.00	93972.60
90	127390.60	128881.00
100	164913.40	168787.80

Table 3.4: The total number of node-to-node key comparison between the trinomial and the quaternary heaps. In this experiment, the concept of adaptive cache is used in the quaternary heap

When the insertion cache was introduced in the quaternary heap, the quaternary heap gave better results. This heap takes advantage of having an ascending order of the input stream by creating a complete trunk in the adaptive cache. For an adaptive cache, the key value of the new inserted node will be compared only with the key value of the node that is located at the end of the trunk. Thus, only one key comparison is needed. When $n < 80$, this data structure is able to perform better than the trinomial heap. The results of the quaternary heap with some modification of the insertion process are shown in Table 3.4.

Even though many complete trunks can be created in the adaptive cache and later will be merged with the existing $\mathbf{T}(1)$ in the heap, the number of key comparisons is still higher in the quaternary heap. These results show that at one point, when n reaches a certain limit, the simple adaptive cache concept can not help in reducing the number of key comparisons in the quaternary heap.

When research into quaternary heap began, the researchers had strong feelings that sparse digraphs were not suitable to use with the quaternary heap. This was mainly because of the structure of the quaternary heap itself, which allowed more nodes on a trunk. The more nodes that the trunk had, the more key comparisons were needed to insert a new node in the heap. Some experiments were done to confirm that the quaternary heap was not suitable to use with sparse digraphs. Table 3.5 shows the results obtained when the sparse digraphs have been used.

Input Size, n	**Trinomial Heap** ($\times 10^3$)	**Quaternary Heap** ($\times 10^3$)
10	359.60	369.40
20	2739.40	2890.20
30	8369.00	8504.00
40	17191.40	19676.80
50	30310.00	33111.00
60	46691.00	50788.40
70	68498.40	77323.20
80	93571.00	105665.00
90	127390.60	141041.20
100	164913.40	176990.00

Table 3.5: The total number of node-to-node key comparison between the trinomial and the quaternary heaps using sparse digraphs

If the dense structure can not give promising results, is there any chance that a very sparse structure can help reducing the number of key comparisons? With that question in mind, the dimensional heap has been developed and the results are shown in the following section.

3.8.2 The performance of the dimensional heap

Experiments have been carried out to see the performance of the new invented heap with well known existing heaps such as binary (28), Fibonacci (3) and 2-3 (4) heaps. Binary heap was used because it is easy to understand and code. Fibonacci heap was chosen because together with the binary heap; these heaps often act as benchmarks to compare with other data structures. It is important to see the performance of these heaps as it is said that binary heaps usually outperform Fibonacci heaps (45). Fibonacci heap is also the first data structure that uses amortized cost in analyzing its performance. It is also interesting to compare the performance of the 2-3 heap as this heap is rarely used even though practically, it shows better performance than the Fibonacci heap (4).

These experiments traced different heaps calls from Dijkstra's shortest-path algorithm. For this experiment, Dijkstra's algorithm was run on three kinds of digraphs: dense, sparse and acyclic digraphs. All heaps were implemented using a linked list for standardization. The same programming style was used to program all heaps.

First, the results show the number of key comparisons when sparse digraphs were used. Table 3.6 shows the results. The results demonstrate that 2-3 heap performs best among other heaps. In this experiment, each of vertex, $v \in V$ (V is the set of vertices) had been assigned to have four outgoing edges ($edge\_factor = 4$). The edge costs were randomly generated.

Input Size, n	**Binary** ($\times 10^3$)	**Fibonacci** ($\times 10^3$)	**2-3** ($\times 10^3$)	**Dheap** ($\times 10^3$)
2000	22.18	24.32	17.55	25.23
4000	48.320	54.81	37.84	56.68
6000	75.97	87.51	58.46	90.54
8000	104.64	121.73	80.07	125.83
10000	133.88	156.89	102.01	162.23

Table 3.6: The total number of node-to-node key comparison between heaps using sparse graphs

In solving the single source shortest path problem (SSSP), Dijkstra's algorithm requires operations such as insert, delete-min and decrease-key operations. In a graph that has n number of vertices and m number of edges, Dijkstra can solve the SSSP in $O(m + n \log n)$ time. Here n delete-min and insert operations are called and m decrease-key operations are required.

The dimensional heap does not give a good performance when the sparse graph is used. The decrease key operation is rarely called by Dijksta's algorithm. The results are satisfied with the finding in (45) that observes that decrease-key function does not appear to be called many times to update the edge cost when a sparse graph is used.

When dense graphs are used, the result is shown in Table 3.7. The units numbers of node-to-node key comparisons.

In Table 3.7, the performance of the dimensional heap is improved. The results show that dimensional heap clearly outperform Fibonacci and 2-3 heaps. It was also quite interesting to see the good performance shown by the binary heap. As the problem size increases, it is said that binary heap is supposed to grow super linearly, thus, other data structures can easily beat the performance of this heap (45).

Input Size, n	**Binary** $(\times 10^3)$	**Fibonacci** $(\times 10^3)$	**2-3** $(\times 10^3)$	**Dheap** $(\times 10^3)$
2000	46.73	43.90	39.28	41.78
4000	83.24	95.08	87.98	89.39
6000	114.66	148.92	156.87	139.90
8000	141.28	203.45	231.94	189.80
10000	174.11	260.14	301.65	244.21

Table 3.7: The total number of node-to-node key comparison between heaps using dense graphs

In dense graphs with n number of vertices, the number of edges, m is given by $m = n^2$. The more edges the graph has, the more keys updates are required. Therefore, using a dense graph can promote the use of the decrease-key function as it will update the key values. Even though the decrease key function is called more frequent when the dense graph is used compared to the sparse graph, this function is still called less than m times. The use of the decrease key function when solving the shortest path problem is not fully optimized.

As the dimensional heap gives special ways to perform the decrease operation, the performance of this process should be explored. If only a normal digraph such as a dense or a sparse graph is used where the edge costs are randomly generated, the performance of the decrease-key operation can not be seen. To see the performance of the decrease-key operation in solving the shortest path problem, a special type of graph should be used. The objective of this experiment is to see that when the decrease-key operations are executed incredibly frequent, as many as m times, which data structure is the best?

Towards that purpose, acyclic digraphs were used in this experiment. The edge costs were specifically given for each (u, v), where $(u, v) \in E$ (E denotes the set of vertices in a graph). Let u be the source vertex, v be the destination vertex and n be the total number of vertices in a graph, g. The below algorithm was used to generate the edge cost for each edge in the graph.

```
procedure CREATE EDGES($g$)
    for $u = 0$ and $u < n$ do
        for $v = u + 1$ and $v < n$ do
            $cost(u, v) = (v - 1) \times (n - u)$;
            increase $v$ by one;
        increase $u$ by one;
end procedure
```

An example of the acyclic graph that force to perform m-decrease-key operation is shown in Figure 3.25.

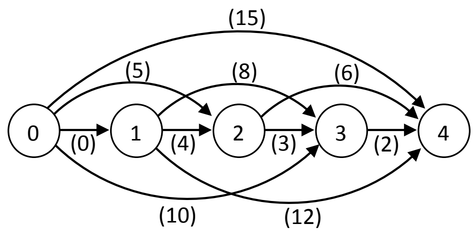


Figure 3.25: An example of an acyclic graph that has five vertices

When running Dijkstra's algorithm with the acyclic graphs, the outstanding result obtained as stated in Table 3.8.

Input Size, n	**Binary** ($\times 10^3$)	**Fibonacci** ($\times 10^3$)	**2-3** ($\times 10^3$)	**Dheap** ($\times 10^3$)
2000	22.95	17.30	16.40	14.35
4000	49.97	35.90	36.71	32.88
6000	78.55	56.83	56.71	47.42
8000	108.14	72.90	75.47	64.95
10000	138.33	93.35	97.68	84.21

Table 3.8: The total number of node-to-node key comparison between heaps using acyclic graphs

The results show that the dimensional heap performs exceptionally well when the acyclic digraphs are used. In fact, the new invented heap required less number of key comparisons than other tested heaps.

3.8.3 Concluding Remarks

The quaternary heap outperforms the trinomial heap when the total number of vertices, n is small enough. However, when n grows, trinomial heap shows better performance. If there is an option to choose a heap, quaternary heap should be considered when only for a small number of problem size. The decrease-key function plays a very important role when comparing the data structures as this operation is very expensive in most of data structures. In a dimensional heap, the decrease-key function is a special function. When m decrease key function is called in solving the shortest path problem, dimensional heap shows outstanding results, and is therefore one of the best options for the data structure.

4

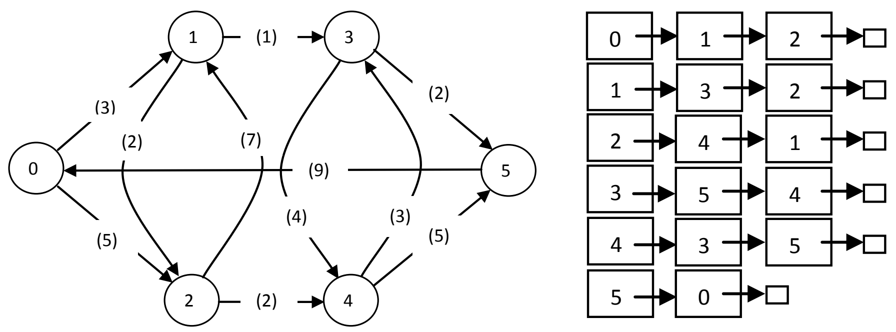
An $O(n^2 \log n)$ Expected Time Algorithm

In this chapter, all pairs shortest path (APSP) algorithms for the average case analysis are explored. The expected running time to solve the APSP in this area is $O(n^2 \log n)$ by the Moffat-Takaoka (MT) algorithm. For solving an APSP, a weighted digraph with edge weights drawn from a random probability distribution is used. For an introduction, this Chapter will discuss a few algorithms that use various techniques for solving the APSP. The existing MT algorithm has been simplified and modified for better analysis. The purpose of this chapter is to show that a small modification of the MT algorithm can achieve the optimal complexity of $O(n^2 \log n)$ with a simpler analysis. To accomplish this, a new algorithm has been developed which is simpler than the MT algorithm. Throughout this Chapter, analyses will be carried out based on the average case analysis that uses complete dense digraphs.

4.1 Introduction

The all pairs shortest path (APSP) can be solved using n single source shortest path (SSSP) problems. Consider the problem of finding the APSP that is represented as a graph. Let $G = (V, E)$ be a directed graph with non-negative edge costs with no self-loop. Here, V and E are the sets of vertices and edges such that $|V| = n$ and $|E| = m$. Labelled vertices are vertices for which the shortest distances from a source s are already known. These vertices are kept in a solution set, S. The cost of edge (u, v) is given by $c(u, v)$. The cost of a path is the sum of the costs of edges that form the path. The shortest path from u to v is the path with the minimum cost. The path cost of v through u is given by $d[v] = d[u] + c(u, v)$. If there is any value of $d[v]$, then $d[v]$ refers to the shortest path cost from s to v that has been found sor far. Initially, $d[s] = 0$ and for all other $v \in V, d[v] = \infty$.

The edges from each vertex v are sorted in non-decreasing order of edge costs. This process is called *pre-sort* or *pre-processing.* A pointer is maintained for the sorted list. The sorted edge list from each vertex v is maintained by putting the endpoints of the sorted edges from v. The example of pre-sort edge list for a sparse graph in Figure 4.1(a) is shown in Figure 4.1(b). If a dense graph is used, $O(n^2 \log n)$ is required to sort the edge lists for the single source problem. A pre-sort is done only once, and the effort used for sorting can be shared over all sources.



(a) A simple graph with $n = 5$

(b) Non-decreasing pre-sort edge list for 4.1(a)

Figure 4.1: A simple sparse graph with its non decreasing pre-sort edge list

For each vertex $u \in S$, a *candidate* is maintained. A candidate, $ce(u)$, is defined as the endpoint of the shortest edge from u. It is said to be *clean* if $ce(u) \notin S$, and non-clean otherwise. A pointer, $P(u)$ is used to point to the current $ce(u)$ and it moves from pointing to the $ce(u)$ to another endpoint, which defines the next current edge, or simply the next $ce(u)$. The function "next of $ce(v)$" is to advance the pointer, $P[v]$, by one and takes the $P[v]$-th member in the list. A set F is used as a frontier set that contains candidates of $u \in S$. The vertices in F will later be chosen to be included in S. Some algorithms that will be discussed here require $L(v)$ that defines a list of vertices that have v as their candidates. For all $u \in L[v]$, u must already be in S.

A *time stamp* concept will also be used in this chapter. A time stamp of v, $T[v]$ denotes the stage when v is included in S. At the beginning, the size of the solution set, $|S|$ is zero. When the first vertex v is inserted in S, then $|S| = 1$. Thus, $T[v] = 1$. In other words, if

$|S| = j,\ T[v] = j$. The basic idea of expanding S for all algorithms explained here is relatively similar to Dijkstra's algorithm (6). To solve the all pair shortest path problem, n single source algorithm is used. For ensuring algorithm efficiency, the implementation of the shortest path algorithms was facilitated by a binary heap (28). A generic algorithm to solve an SSSP is given in Algorithm 4.

Algorithm 4 A generic algorithm to solve SSSP

```
1:  procedure SINGLE_SOURCE($n$)
2:      for $v \in V$ do $d[v] = \infty$;
3:      $ce(s)$ = next of $ce(s)$; $t = ce(s)$;
4:      $d[s] = 0;\ d[t] = c(s,t);\ F = \{s\}$;
5:      organize $F$ in a priority queue with $d[t]$ as key;
6:      $S = \emptyset$;
7:      while $|S| < n$ do
8:          find $u$ in $F$ with minimum key;                    ▷ find-min
9:          $v = ce(u)$;
10:         if $v \notin S$ then
11:             $S = S \cup \{v\}$;
12:             UPDATE($v$);
13:         UPDATE($u$);
14: end
15: procedure UPDATE($v$)
16:     perform some scanning by increasing $P(v)$;
17:     let $w = ce(v)$;
18:     $d[w] = \min\ \{d[w], d[w] + c(v,w)\}$;
19:     $key(v) = d[v] + c(v,w)$;
20:     if $v \in F$ then
21:         increase-key($v$);                    ▷ increase-key $v$ with $key(v)$
22:     else
23:         insert($v$) with $key(v)$; $F = F \cup \{v\}$;        ▷ insert($v$)
24:     reorganize $F$ into the heap with new $key(v)$;
25: end
```

Algorithm 4 works as follows: First, a vertex u that has the minimum key value is selected from the priority queueor sometimes heap is used here. Then, a candidate, v of $u, ce(u)$ is obtained. If v is not a member of the solution set, S, it is inserted to S and $d[v]$ is set to be the final distance cost, or the shortest path from the source, s. Later, *update* procedure is called to update v, followed by updating u.

The *update(v)* procedure is called to update v with a new candidate and its key value. A pointer of $v, P(v)$ that points to its candidate will be reviewed. Let w be the current candidate of v, pointed to by $P(v)$. The path cost from v to w is obtained, and if the cost is smaller than the existing $d[w]$, the value of $d[w]$ will be updated with $d[w] = d[v] + c(v,w)$, where $c(v,w)$ is the edge cost from v to w. Lastly, if v is already in the heap, an *increase-key* function is called to increase the key value of v with the new key, $key(v) = d[v] + c(v, w)$. If w is not in F, it is inserted into the heap with the above key value.

Selecting the next candidate varies from one algorithm to another. To select the candidate, a scanning process is performed. Consider the generic algorithm to solve a single source problem, as shown in Algorithm 4. In the *update(v)* procedure, a pointer $P(v)$ is used to find the candidate that is located in the pre-sort array. Some algorithms require $P(v)$ to move only one step ahead, some demand $P(v)$ to move until a clean candidate is found, and some other algorithms are flexible by asking $P(v)$ to move a certain number of steps according to some criterion. Note that the movement of $P[v]$ for all v in V is represented in the algorithm as $ce(v) =$ next of $ce(v)$.

In the following section, various scanning techniques used in finding the next candidate are explored. The techniques used vary from one algorithm to another; thus, each has significantly different performance parameters. The different techniques used are divided into the following three main categories: unlimited scanning, simple scanning, and limited scanning.

4.2 Unlimited Scanning Algorithms

The scanning process on vertex v is a routine to find and select a new candidate of $v, ce(v)$. The term unlimited scanning comes from the fact that scanning is done repeatedly, with no limit until the required clean candidate is found. The first and foremost algorithm to discuss is Dantzig's algorithm (46).

Dantzig's algorithm only allows clean candidates to be chosen. That means, for each $u \in S, ce(u)$ must be a vertex that has not been included in the solution set yet. To find v, the *update* procedure as shown in Algorithm 4 is modified as follows. Let w be the candidate of v. If w is already in S, increase $P(v)$ by one and a new w is checked. If the new w is also in S, $P(v)$ will move to the next edge again. This process is a repeated process until the last w is guaranteed to be a clean one.

To describe Dantzig's in further detail, let vertex u in S have a candidate v. The distance or key value of u is defined by the distance from s to u plus the edge cost of (u, v), where s is the source vertex. The candidate v might also be the candidates of other u in S with some distance values. Those u are kept in the list $L[v]$. To expand S, the candidate v of u with the smallest key value is chosen and labelled. When the candidate v is included in S, no other

Algorithm 5 Dantzig's algorithm to solve SSSP

```
1: procedure SINGLE_SOURCE($n$)
2:     for $v \in V$ do $d[v] = \infty$;;
3:     $t = ce(s); ce(s) =$ next of $ce(s)$;                    ▷ $P[s]$ increases by one
4:     $d[s] = 0$;  $d[t] = c(s,t)$;  $F = \{s\}$;
5:     organize $F$ in a priority queue with $d[t]$ as key;
6:     $S = \emptyset$;
7:     while $|S| < n$ do
8:         find $u$ in $F$ with minimum key;                    ▷ find-min
9:         $v = ce(u)$;
10:        if $v \notin S$ then
11:            $S = S \cup \{v\}$;
12:            DANTZIG'S_UPDATE($v$);
13:        for $u \in L[v]$ do
14:            DANTZIG'S_UPDATE($u$);                ▷ update all incoming edges to $v$
15: end
16: procedure DANTZIG'S_UPDATE($v$)
17:     let $w = ce(v)$;
18:     while $w \in S$ do                                      ▷ scanning effort
19:         $ce(v) =$ next of $ce(v)$;
20:         $w = ce(v)$;
21:     $d[w] = \min\{d[w], d[w] + c(v,w)\}$;
22:     $L[w] = L[w] \cup \{v\}$;                              ▷ append $v$ to $L[w]$
23:     $key(v) = d[v] + c(v,w)$;
24:     if $v \in F$ then
25:         increase-key($v$);                   ▷ increase-key $v$ with $key(v)$
26:     else
27:         insert($v$) with $key(v)$;  $F = F \cup \{v\}$;        ▷ insert($v$)
28:     reorganize $F$ into the heap with new $key(v)$;
29: end
```

vertex can possibly choose it as a candidate. Therefore, other vertices whose candidates have just been labelled, that is, $u \in L[v]$, need to be revised with new candidates. The process is repeated until all vertices are labelled. Figure 4.2 shows an example of a stage in Dantzig's algorithm. Dantzig's algorithm is shown in Algorithm 5.

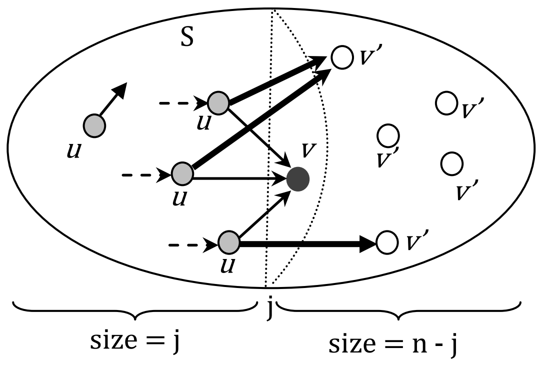


Figure 4.2: Some intermediate stage during the expansion of S in Dantzig's algorithm

In Dantzig's algorithm, the candidate v is always a clean one. Choosing only a clean candidate for each u in S requires significant effort, as it is essential to perform unlimited scanning of the edge list to find this candidate. By doing detailed scanning, the minimum weight candidate in the heap is guaranteed to be unlabelled vertex, which can be included in S. In this algorithm, the expansion of S is clearly proven to be $O(n)$ time but the scanning effort used to scan for a clean candidate is very expensive. When $|S| = j$, $O(j)$ effort is required to search for a clean candidate, totalling $O(n^2)$ efforts are needed for n number of vertices. The cost to do pre-sort of edges is $O(n^2 \log n)$ time. Therefore, to solve the single source shortest path(SSSP) problem, Dantzig's requires $O(n^2 + n^2 \log n)$ time and $O(n^3)$ time for the all-pairs shortest path problem(APSP).

When the Dantzig algorithm was first implemented, no priority queue was used. This algorithm was also designed to solve a single source shortest path problem. However, as the cost of pre-sorting itself is $O(n^2 \log n)$ time for a dense graph, this algorithm is best practiced to solve the APSP problem as the cost of pre-sorting can be absorbed in $O(n^3)$.

4.3 Simple Scanning by One

Here, Spira's algorithm(29) is described. In Spira's algorithm, a priority queue is proposed to facilitate a few operations, such as finding and deleting the minimum key and updating key values. Spira also applied the same ideas as Dantzig. The edges from each vertex $v \in V$ are sorted in non-decreasing order that takes $O(n^2 \log n)$ time for a complete n vertices graph. A pointer is also maintained for the sorted list. A pointer $P(v)$ always points to the current edge and it will be moved by one in *update* to get to the next edge.

Spira's algorithm maintains the solution set, denoted by S, which is the set of vertices to which shortest paths have so far been established by the algorithm, in a priority queue Q. The key for u in the queue, $key(u)$, is given by $key(u) = d[u] + c(u, ce(u))$, where $d[u]$ is the known shortest distance from the source to u.

Compared to Dantzig's, Spira's allows a candidate of $u \in S$ to be in S, which is a non-clean candidate. To expand S, this algorithm works similarly to Dantzig's but does not require u to be updated with the new unlabelled candidate.

The queue is initialized with one element of s, the source. Let $key(s) = c(s, t)$, where edge(s, t) is the shortest edge from s. Obviously t is included in the solution set as the second member. In general, suppose u is the minimum of the queue, that is, $key(u)$ is minimum in the queue. If $v = ce(u)$ is not in S, it can be included in S with $d[v] = key(u)$, and then included in Q with $key(v) = d[v] + c(v, w)$, where (v, w) is the shortest edge from v.

Regardless of whether the above v is in S or not, the pointer on the edge list from u is advanced to the next element because edge (u, v) is no longer useful, which means that this edge is not going to be examined for other shortest paths.

The priority queue Q needs to support *find-min, increase-key* and *insert* operations efficiently, which is expressed by the repertory (*find-min, increase-key, insert*). Spira used a tournament tree for a priority queue in his algorithm, which supports the first operation in $O(1)$ time and the last two operations in $O(\log n)$ time. In this thesis, a more common data structure is used, ordinary binary heap, which supports the same set of operations with the same time complexity. All pointers for edge lists are initialized to 0. To point to the first member in the edge list, $P[v] = 1$. The sorted list of edges for each vertex starts from index 1. The algorithm for the single source problem follows.

Figure 4.3 shows the expansion of the solution set, S at the j-th stage.

Algorithm 6 Spira's algorithm to solve SSSP problem

```
 1: procedure SINGLE_SOURCE($n$)
 2:     for $v \in V$ do   $d[v] = \infty$;
 3:     $ce(s)$ = next of $ce(s); t = ce(s)$ ;                    ▷ $t$ is the first candidate of $s$
 4:     $d[s] = 0$ ; $F = \{s\}$ ; $d[t] = c(s,t)$;
 5:     organise $F$ in a priority with $key(s) = c(s,t)$;
 6:     $S = \emptyset$;
 7:     while $|S| < n$  do
 8:         find $u$ in $F$ with minimum key;
 9:         $v = ce(u)$;
10:         if  $v \notin S$ then
11:             $S = S \cup \{v\}$;
12:             SPIRA_UPDATE($v$);
13:         SPIRA_UPDATE($u$);
14: end
15: procedure SPIRA_UPDATE($v$)
16:     $ce(v)$ = next of $ce(v)$,                                  ▷ scanning effort
17:     $w = ce(v)$;
18:     $d[w] = \min\ \{d[w], d[v] + c(v,w)\}$;
19:     $key(v) = d[v] + c(v,w)$;
20:     if $v$ is in a heap then
21:         increase-key($v$);                          ▷ increase-key $v$ with $key(v)$
22:     else
23:         insert($v$); $F = F \cup \{v\}$;                              ▷ insert($v$)
24:     reorganize $F$ into the heap with new $key(v)$;
25: end
```

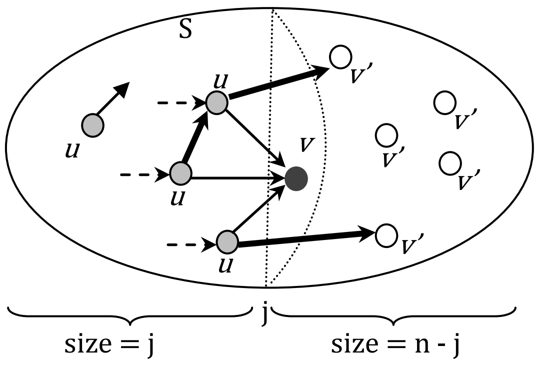


Figure 4.3: Some intermediate stage during the expansion of S in Spira's algorithm

For the analysis, the endpoint independence model is used for the probabilistic assumption. In this model, when the edge list is scanned, any vertex appears independently with a probability of $\frac{1}{n}$. When there are less than n edges, it is assumed that edges with costs of infinity, randomly and independently attached at the end of the list. This model was chosen as it is commonly used for the average case analysis.

Let $U = (T_1, \ldots, T_{n-1})$ be the times for expanding the solution set by one at each stage of the size. Let E_X be the expectation operator over the sample space of random variable(s) X. Then, ignoring some overhead time between expansion processes, the expected value $E_U[T]$ of the total time $T = T_1, \ldots, T_{n-1}$

$$E_U[T] = E_U[T_1 + \ldots + T_{n-1}] = E_U[T_1] + \ldots + E_U[T_{n-1}].$$

From the theorem of total expectation, $E_Y[E_X[X|Y]] = E_{X,Y}[X]$, where $X|Y$ is the conditional random variable of X conditioned by Y. In this analysis, X represents a particular T_i, Y is for the rest and (X, Y) for U. The fact that $E_X[X|Y]$ is the same for all Y is used from the endpoint independence. This idea enables us to localize analysis in each stage of expansion, and will be used in later sections for various analyses.

To analyze Spira's, let T_j represent the expansion of the solution set, S from size j to $j+1$ where $|S| = j$. At the j-th stage, the heap contains j candidates. The probability that v is outside S at line 10 is $\frac{n-j}{n}$ from the endpoint independence. The number of executions of *find-min* at line 8 is given by the reciprocal of this probability; that is, $\frac{n}{n-j}$, which corresponds to the above $E_X[E[X|Y]]$. Note that $E_X[X|Y] = E_Y[X|Y]$ in this scenario, since $\frac{n}{n-j}$ does not depend on Y, that is other T_i's for $i \neq j$.

Each time when the *find-min* is executed, $O(1)$ time is spent in finding the minimum node and $O(\log n)$ time in *update* at line 13. Thus from the above total expectation, the expected time for line 8 and 13 is:

$$n \log n \sum_{j=1}^{n-1} \frac{1}{n-j} = O(n \log^2 n)$$

The *update* at line 12 is executed exactly $n-1$ times. Thus a separate analysis can give us $O(n \log n)$ time, which is absorbed into the above main complexities. The APSP takes $O(n^2 \log^2 n)$time. In this thesis, the base for logarithm is not specified, as the quantities are equivalent within O-notation.

Spira's algorithm is inefficient as it allows u with a non-clean candidate to be in the heap with a relatively large probability, and later to be chosen. The running time is increased for this uneconomical operation. In the following section, the scanning method is ameliorated to improve the probability that the candidates are clean by some limited scanning.

4.4 Limited Scanning Algorithms

As it is important to only scan for clean candidates to reduce time spent on the expansion of S, a few algorithms have been implemented using a concept of limited scanning. These algorithms attempt to overcome the inefficiency of algorithms 4.2 and 4.3, by attempting to search for a clean candidate in certain time limitation. There are two different ways to do this: the movement of pointer is limited up to m particular times; another is to use a timestamp concept. The timestamp, $T[v]$ refers to the stage when vertex v is included in S. To prepare for the later explanation, the $T[v]$ can be regarded as the time stamp of v. For v in $S, 1 \leq T[v] \leq j$, where j is the size of S. When a source vertex s is included in S, the value of j is one. Hence, $T[s] = 1$. At j-th stage, vertex v is inserted, thus, $T[v] = j$. The details of algorithms under this category will be explained further below.

4.4.1 Limited scanning up to a fixed number of times algorithms

The candidate $ce(u)$ of u is clean if it is outside S, and non-clean, otherwise. In Spira's algorithm, when the next edge from the edge list in *update* is chosen,the new candidate may be non-clean. It may be expensive to scan the edge list until a clean candidate if found as in (46). However a careful design of scanning strategy may bring down the complexity.

To avoid a potentially long scanning time, Bloniarz(30) introduced the idea of limited edge scanning for a clean candidate. The technique leads to asymptotically improved running time as it makes the probability to get a clean candidate higher. Bloniarz not only proposed an effective scanning technique; it is also effectively free. The free concept is used to describe that the operation is completely absorbed by other operations.

Let m be the number of times that the pointer is increased. The pointer of $v, P(v)$ is increased until a clean candidate is found or m reaches $\lceil \log n \rceil$ edges. If the clean candidate is obtained during the scanning process, it will selected as the next candidate. Otherwise, if $m \geq \lceil \log n \rceil$, the selected candidate is allowed to be a non-clean one. Bloniarz's algorithm improves the running time over Spira's algorithm by trying to avoid choosing a vertex in S. Bloniarz's *update* procedure is shown below.

```
1:  procedure BLONIARZ'S UPDATE($v$)
2:      $counter = 0$; // $counter = m$
3:      let $w = ce(v)$;
4:      while $w \in S$ and $counter \leq \lceil \log n \rceil$ do          ▷ scanning effort
5:          $ce(v)$ = next of $ce(v)$;
6:          $w = ce(v)$;
7:      $d[w] = \min \{d[w], d[w] + c(v, w)\}$;
8:      $key(v) = d[v] + c(v, w)$;
9:      if $v \in F$ then
10:         increase-key($v$);                          ▷ increase-key $v$ with $key(v)$
11:     else
12:         insert($v$) with $key(v)$; $F = F \cup \{v\}$;              ▷ insert($v$)
13:     reorganize $F$ into the heap with new $key(v)$;
14: end
```

This algorithm solves the all pairs shortest path problem (APSP) with expected time $O(n^2 \log n \log^* n)$. Under suitable probability distributions and implementations restrictions, Bloniarz's obtains $\Omega(n \log n)$ time for lower bound in the worst case and $O(n \log n \log^* n)$ upper bound in the average case to solve the single source shortest path(SSSP) (47).

The next algorithm is Takaoka-Moffat's algorithm (47). This algorithm is very comparable to Bloniarz's method; it uses a hybrid technique of scanning, that is, the m scanning time is applied, with the usage of timestamp idea. Similar to Bloniarz's, Takaoka-Moffat's algorithm attempts to locate a clean candidate, by moving a pointer of $v, P(v)$ over the sorted edge list. This process ends when a clean candidate is found or when the total count m to move the pointer is greater than $\frac{n}{n-T[v]}$, regardless whether the candidate is clean or not. The total frequency, f, that is, the number of *delete-mins* is proved in (47) to be $f = \sum_{j=1}^{n-1} \frac{1}{p_j} = 2n \log_e \log_e n + const$. Thus, the total running time to perform heap operations is $O(n \log n \log \log n)$ as *increase-key* procedure takes $O(\log n)$ time. The *find-min* is done in $O(1)$ time. Another important factor to consider is scanning effort used to find clean candidates. The scanning effort to scan a clean candidate is $\frac{1}{p_j} \sum_{i=1}^{j} \frac{1}{j} . \frac{n}{n-i}$ as t can be any integer from 1 to j with probability $\frac{1}{j}$. The scanning effort above is shown to be $O(n \log n)$ time, which is absorbed in the main complexity.

```
1: procedure TAKAOKA-MOFFAT'S UPDATE($v$)
2:     $counter = 0$;
3:     let $w = ce(v)$;
4:     while $w \in S$ and $counter \leq \frac{n}{n-T[v]}$ do                ▷ scanning effort
5:         $ce(v)$ = next of $ce(v)$;
6:         $w = ce(v)$;
7:     $d[w] = \min \{d[w], d[w] + c(v, w)\}$;
8:     $key(v) = d[v] + c(v, w)$;
9:     if $v \in F$ then
10:        increase-key($v$);                               ▷ increase-key $v$ with $key(v)$
11:    else
12:        insert($v$) with $key(v)$; $F = F \cup \{v\}$;                     ▷ insert($v$)
13:    reorganize $F$ into the heap with new $key(v)$;
14: end
```

The above algorithms were implemented by using the same strategy; they start by initializing a source vertex s and expand S by inserting the shortest path from s. Some algorithms are superior in the total time of heap operations while others have better running time in scanning effort. It is known that the crucial point in any APSP algorithms analysis is to measure the total number of comparisons of all operations in the heap and the scanning effort to get clean candidates. Unlimited scanning effort was introduced in Dantzigs algorithm, resulting in time for a single source problem in $O(n^2)$ (46). Spira ignores the concept of searching for good candidates, which eases scanning effort while increasing the total number of comparisons.

Limiting the scanning efforts to balance with the time spent for expanding seems to be the best strategy so far to solve the shortest path problems as in (30) (47) (1).

4.4.2 Timestamp Scanning

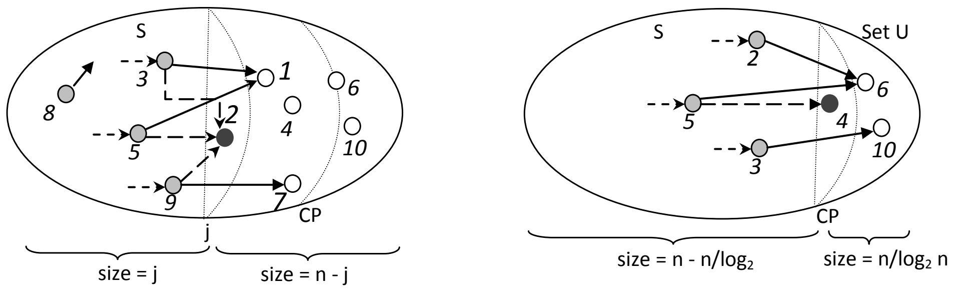
The Moffat-Takaoka(MT)(1) algorithm solves the APSP problem in $O(n^2 \log n)$ by dividing the expansion of S into two phases. This blended technique uses Dantzig's algorithm for the expansion of S in the first phase, followed by Spira's algorithm in the second phase. When $|S| \le n - \frac{n}{\log n}$, the algorithm is said to be in the first phase and the second phase otherwise. These phases are divided by a critical point CP the moment when the size of the solution set is equal to $|S| = n - \frac{n}{\log n}$. Algorithm 7 shows the implementation of MT algorithm.

Algorithm 7 The original MT's algorithm to solve the SSSP problem.

```
1: for $v \in V$ do $P(v) = 0$;
2: $t = ce(s)$ ;
3: $S = \{s\}$;
4: $d[s] = 0$ ; $F = \{s\}$ ;
5: $d[t] = d[s] + c(s,t)$;
6: organise $F$ in a priority with $key(s) = c(s,t)$;
7: while $|S| \le n - \frac{n}{\log n}$ do                          ▷ first phase
8:     follow Dantzig's algorithm
9: end
10: re-initialize a heap with keys in $U$.
11: while $|S| > n - \frac{n}{\log n}$ do                         ▷ second phase
12:     follow Spira's algorithm
13: end
```

This algorithm performs an unlimited search for clean candidates before the critical point, and a limited search after the critical point. To identify the critical point, an array element $T[v]$ is maintained, which gives the order in which v is included in S, and is called the time stamp of v. Like Spira's algorithm, members of S are organized in a binary heap. The time for heap operations is measured by the number of (key) comparisons. As in Algorithm 6, all pointers for edge lists are initialized as 0.

Initially, each vertex $v \in S$ has a candidate $ce(v)$, which the endpoint vertex of the shortest edge from v. Before the critical point, CP, each candidate of $v \in S$ is required to be only a clean one; it means $ce(v)$ should be located outside the current S. This can be done by scanning the sorted list of v's endpoints until a clean vertex is found. However, after the CP, $ce(v)$ can be a non-clean candidate. Let $U = V - S$, that is, $|U| = \frac{n}{\log n}$. In the second phase, only U-vertices, that is, vertices $v \in U$, are used as candidates and inserted into a heap. The expansion of S in MT algorithm is shown in Figure 4.4.



(a) One intermediate stage in Phase 1   (b) One intermediate stage in Phase 2

Figure 4.4: Some intermediate stage during the expansion of S in MT's algorithm

In (48), the MT algorithm is simplified as shown in Algorithm 8. The critical point is maintained as $CP = n - \frac{n}{\log n}$. However, the timestamp concept is used to split edges into two phases. When $T[w] \leq n - \frac{n}{\log n}$, that means candidate w is said to be included in S in the first phase. If $T[w] > n - \frac{n}{\log n}$, candidate w is in the U set.

The list $L[v]$, called the batch list, for each vertex v, whose members are vertices u such that $ce(u) = v$ is maintained. The key for vertex u in the priority queue, $key(u)$, is given by $key(u) = d[u] + c(u, ce(u))$. Whether v is found to be a member of S at line 12 or not, those members in $L[v]$ need to be updated at line 22 to have more promising candidates. Also v itself needs to be treated to have a reasonable candidate at line 18 when v is included in S. How much scanning needs to be done for a good candidate is the major problem hereafter.

Computing time consists of two major components. One is the number of key comparisons in the heap operations and the other is the time for the scanning effort on the edge lists. The times before CP and after CP are both $O(n \log n)$ and balanced in both comparisons and scanning. If the limit is set to infinity for all computations, an unlimited search for clean candidates is carried out, and the resulting algorithm is Dantzig's algorithm (46), which is more expensive.

Before CP, all candidates are clean, meaning that the expansion of S from j to $j+1$ is done with probability 1 at line 12 and $O(n \log n)$ heap operations are done in total. Scanning effort to go outside S is $O(\log n)$ before CP, resulting in $O(n \log n)$ time.

Let $U = V - S$ when $|S| = n - \frac{n}{\log n}$, that is, $|U| = \frac{n}{\log n}$. Before CP all candidates are clean, meaning the if-condition at line 12 is satisfied with probability 1 and $O(n \log n)$ time is spent

Algorithm 8 A revised MT algorithm to solve the SSSP problem.

```
procedure SINGLE_SOURCE($n$)
    for $v \in V$ do $T[v] = \infty$;
    $t = ce(s)$;
    $j = 1$; $S = \{s\}$; $T[s] = 1$;
    $ce(s) =$ next of $ce(s)$;                          ▷ $P[s]$ increases by one
    $d[s] = 0$ ; $F = \{s\}$ ;
    $d[t] = c(s,t)$;
    organise $F$ in a priority with $key(s) = c(s,t)$;
    while $|S| < n$ do
        find $u_0$ in $F$ with minimum key;
        $v = ce(u_0)$;
        if $v \notin S$ then
            $S = S \cup \{v\}; j = j + 1$; $T[v] = j$;
            if $j \leq n - \frac{n}{\log n}$ then
                $limit = \infty$;
            else
                $limit = n - \frac{n}{\log n}$;
            UPDATE($v$);
        if $limit < \infty$ then                          ▷ first phase
            for $u \in L[v]$ do
                delete $u$ from $L[v]$;
                UPDATE($u$);                              ▷ $u_0$ is included
        else                                              ▷ second phase
            UPDATE($u_0$);
end
procedure UPDATE($v$)
    $w = ce(v)$;
    while $w \in S$ and $T[w] \leq limit$ do              ▷ scanning effort
        $ce(v) =$ next of $ce(v)$;
        $w = ce(v)$;
    $L[w] = L[w] \cup \{v\}$;                             ▷ append $v$ to $L[w]$
    $d[w] = min\{d[w], d[v] + c(v,w)\}$;
    $key(v) = d[v] + c(v,w)$;
    if $v$ is in a heap then
        increase-key($v$);                                ▷ increase-key $v$ with $key(v)$
    else
        insert($v$); $F = F \cup \{v\}$;                  ▷ insert($v$)
    reorganize $F$ into the heap with new $key(v)$;
end
```

for *insert* operations in total. Scanning effort to get a candidate outside S is $O(\log n)$ before CP, resulting in $O(n \log n)$ time.

In the first phase, candidates of labelled vertices must be clean candidates. Labeling vertices as members in S is modeled as the coupon collector's problem(49). To collect n different coupons means $O(n \log n)$ coupons are needed. After CP, all candidates are limited to U, meaning that the process is modeled as collecting $\frac{n}{\log n}$ coupons. Thus,

$$\begin{aligned}
\frac{n}{\log n} \log \left(\frac{n}{\log n} \right) &= \frac{n}{\log n} (\log n - \log(\log n)) \\
&= \frac{n}{\log n} \left(1 - \frac{\log(\log n)}{\log n} \right) \\
&\leq n \\
&= O(n)
\end{aligned}$$

Therefore, $O(n)$ trials are needed, meaning $O(n \log n)$ comparisons are required. The useful lemma is stated.

Lemma 1 *Let there be a heap of n elements with random keys. If keys of nodes are changed at random with the assumption that the probability that the key of a node be changed is p, then the tree can be restored back into a heap in $O(pn + \log n)$ expected time.*

Proof The results given in (1).

The analysis of *increase-key* in *update* before CP involves some probabilistic analysis on members in the batch list. In (1), vertices u in the batch list $L[v]$ are processed for *increase-key* in a bottom-up fashion, and the time for this is shown to be $O(\log n)$ before CP from Lemma 1. Since p can be substituted as $\frac{1}{n-j}$ and j for n, thus, the summation for the batch processing for the restoration of the heap becomes

$$\sum_{j=1}^{n-1} \left(\frac{j}{n-j} + \log j \right) = O(n \log n)$$

This form $\frac{j}{n-j} + \log j$ remains $O(\log n)$ until the critical point is reached, but exceeds the target complexity after it. To avoid this analysis, (50) uses a Fibonacci heap with (*delete-min, decrease-key, insert*) for maintaining candidates of vertices in a queue. This simplifies analysis for the *update* for $L[v]$, but after CP, the heap must be re-initialized to include S and operations must be switched to (*delete-min, increase-key, insert*).

The scanning effort is not easy to analyze after CP, as the last movement of the pointer at each vertex (called an over-run) does not always lead to successful inclusion of the candidate vertex. In (1) the probabilistic dependence before and after CP regarding the amount of over-run was overlooked, and in (50) an analysis on this part is given, where the over-run associated with each vertex is regarded as a random variable conditioned by the behavior of Spira's algorithm.

This analysis of "over-run" motivates the simplified new algorithm in the next section for a simpler analysis.

4.5 A New Algorithm

The MT algorithm shows that the expected running time to solve the all-pairs shortest path problem(APSP) is $O(n^2 \log n)$. The drawback of this algorithm is only the use of a critical point, which is bumpy and insufficiently smooth in this algorithm. In the first phase, unlimited search is done. When the algorithm enters the second phase, it has to perform a simple scan, means the behaviour of MT algorithm is changed. If this algorithm can be improved by removing the critical point concept, is there any chance that the running time complexity maintained? With this question in mind, a new algorithm has been developed, removing the concept of a critical point, and always make a balance of the total run time complexity during the expansion of the solution set from the beginning towards the end when solving the APSP problem.

The new algorithm devides all vertices into three set of vertices: solution set, buffer zone and new area. These divisions are maintained upon the expansion of S from $j = i$ to $j = i + 1$, where $j = |S|$. The buffer zone is a subset of recent members of the solution with a size $\frac{n}{\log n}$. Vertices v such that $T(v)$ is from $j - \frac{n}{\log n}$ to j are members. When a vertex is put in the solution set, a time stamp is given to it, which is the new size of the solution set. The concept of time stamp used here is similar to the time stamp introduced in the earlier sections. The new area is outside the solution set. The set given by the union of the buffer zone and the new area is called the valid area. For easy explanation of these sets of vertices, Figure 4.5 can be referred to.

In the new algorithm, the candidate vertices for all $u \in S$ are kept in the valid area. The candidates are said to be clean candidates if they are in the new area, but maintaining them there is expensive. The strategy of keeping candidates in the valid area is used, that is, "half-clean" throughout the computation. The new algorithm is seamless, so to speak.

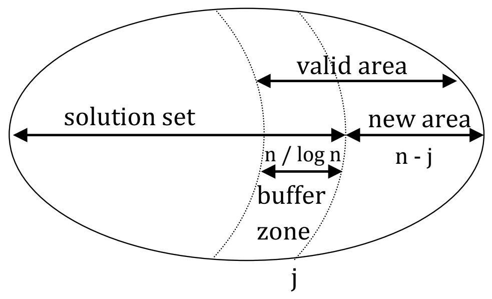


Figure 4.5: The three areas of vertices distribution

In general, this algorithm is also similar to Takaoka-Moffat's and Bloniarz's algorithms which scan the pre-sorted edge lists to find clean candidates. However, during the expansion of S, this algorithm allows candidates for all $u \in S$ to be chosen from the valid area; that means the candidate that will be chosen can be a non-clean one if it is selected from the buffer zone.

The data structure L, called the batch list, is also used and was needed in the MT algorithm for maintaining completely clean candidates before the critical point. This data structure is utilized to keep candidates half-clean, which means candidates are outside the solution set with some probability. For the priority queue, the classical binary heap is used. As Goldberg and Tarjan (51) have pointed out, the binary heap is the best choice from a practical point of view for the implementation of Dijkstra's algorithm, since decrease-key in a Fibonacci heap, which takes $O(1)$ amortized time, is not performed frequently on average. It is shown in the new algorithm's framework as well that the binary heap works well.

The solution set is expanded in the same way as other algorithms by choosing the minimum vertex, u in the heap. The candidate of $u, v = ce(u)$ is identified. If v is not in S, it will be added in S. Here j will be increased by one, thus $T[v] = j$. Then, vertices u and v will be updated with new candidates. The new candidates can only be chosen from the valid area of size $\left((n-j) + \frac{n}{\log n}\right)$. In other words, if the endpoint of the current edges of vertices u and v are outside the valid area, then the pointer should keep moving to find the next candidate in the buffer zone or in the new area. The details of this algorithm are illustrated in the Algorithm 9. Some intermediate stage during the expansion of S is given in Figure 4.6. A primary version of this algorithm can be found in (52).

The amount of scanning can be determined by the bound on pointer movements in (47), (30) and (48), which is called bound-oriented scanning, whereas in (1) and (50) scanning is

Algorithm 9 A new algorithm to solve the SSSP problem.

```
1:  procedure SINGLE_SOURCE($n$)
2:      for $v \in V$ do $T[v] = \infty$;
3:      $t = ce(s)$ ;
4:      $j = 1$; $S = \{s\}$; $T[s] = 1$;;
5:      $ce(s)$ = next of $ce(s)$;
6:      $d[s] = 0$ ; $F = \{s\}$ ;
7:      $d[t] = d[s] + c(s, t)$;
8:      organise $F$ in a priority with $key(s) = c(s, t)$;
9:      while $|S| < n$ do
10:         find $u_0$ in $F$ with minimum key;
11:         $v = ce(u_0)$;
12:         if $v \notin S$ then
13:             $S = S \cup \{v\}; j = j + 1$;
14:             $T[v] = j$;
15:             UPDATE($v$);
16:             for $u \in L[v_0]$ do
17:                 delete $u$ from $L[v_0]$;            ▷ $v_0$ is the expiring vertex
18:                 UPDATE($u$);
19:         delete $u_0$ from $L[v]$;
20:         UPDATE($u_0$);
21: end
22: procedure UPDATE($v$)
23:     $w = ce(v)$;
24:     while $w \in S$ and $T[w] \leq n - \frac{n}{\log n}$ do            ▷ scanning effort
25:         $ce(v)$ = next of $ce(v)$;
26:         $w = ce(v)$;
27:     $L[w] = L[w] \cup \{v\}$;            ▷ append $v$ to $L[w]$
28:     $d[w] = min\{d[w], d[v] + c(v, w)\}$;
29:     $key(v) = d[v] + c(v, w)$;
30:     if $v$ is in a heap then
31:         increase-key($v$);            ▷ increase-key $v$ with $key(v)$
32:     else
33:         insert($v$); $F = F \cup \{v\}$;            ▷ insert($v$)
34:     reorganize $F$ into the heap with new $key(v)$;
35: end
```

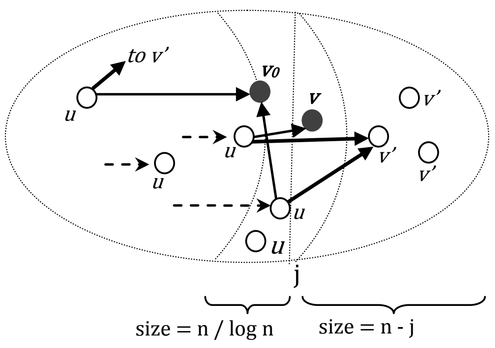


Figure 4.6: Some intermediate stage during the expansion of S in Spira's algorithm

done until a specifed destination is found, which is called destination-oriented scanning. The proposed algorithm belongs to the category of destination oriented search, that is, a one-phase algorithm with destination-oriented scanning. Spira's algorithm that has been explained in the earlier section is a special case in this category with the destination being any set. It also belongs to the former category of bound-oriented where the bound is 1.

At a certain j-th stage, there will be a vertex v_0 that $T[v_0] = j - \frac{n}{\log n}$. This v_0 is on the border of the valid area. It may be called an expiring vertex. Those vertices, u , which are currently pointing to v_0 have to point to the next suitable candidates v' as illustrated in Figure 4.6. This can be seen at lines 16-18 in Algorithm 9. The expiring vertex in depicted in Figure 4.7. The new algorithm, Algorithm 9, does a limited search for clean candidates in the edge

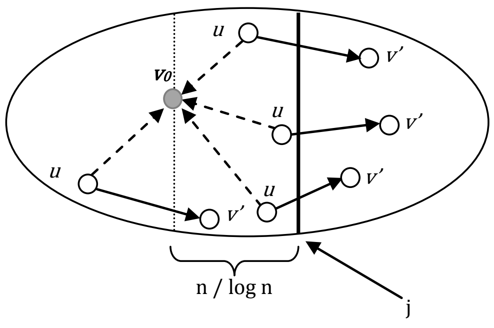


Figure 4.7: Illustration of the expiring vertex v_0 requiring all $u \in S$ to point to the next candidates in the valid area.

list. The target is dynamically changing and given by the set of vertices whose time stamp is greater than $j - N$, where $N = \frac{n}{\log n}$ and j is the size of the current solution set. The size of the valid area is $n - j + N$. The probability to hit the valid area is $\frac{(n-j+N)}{n}$, and the number of pointer movements to hit this area is $\frac{n}{(n-j+N)}$. The fact that $\frac{n}{(n-j+N)} \leq \log n$ for all j is important, as the over-run can be bounded by $O(\log n)$ on average, and need not be analyzed separately. Members of S are organized in a binary heap as in Algorithm 8. The while loop starting from line 9 is the main iteration. Vertex v_0 is the expiring vertex from the valid area, that is, $T[v_0] = j - N$.

The idea of this simplified algorithm is to optimize the choice of selecting a good or clean candidate. To choose only the clean candidate is very expensive and to choose only the next candidate such as Spira algorithm is also not the best practice as it is very pricy to expand the solution set. It is MT algorithm that motivates the establishement of this new algorithm. It is said that the best algorithm should be simple and easy to implement and this is how this algorithm is represented; solving a problem smoothly and steadily from the beginning until the end. Smoothly and steadily mean, there is no concept of changing the behaviour of the algorithm during the expansion of S such as the critical point concept in MT algorithm. In other words, the algorithm does not distinguish between phases. It stops when all vertices have been labelled.

4.5.1 Correctness

The correctness of a generic algorithm with limited scan including algorithms 6, 7, 8 and 9 comes from the following two lemmas borrowed from (30).

Here, limited search means the pointer on the edge list moves until it hits a vertex outside S or goes a certain number of steps according to some criterion of the algorithm. Spira is a special case of limited search. Proof is done by induction following the execution of the algorithm.

Lemma 2 *Assume vertex $v \in S$ is such that $ce(v)$ is not in S and $d[v]+c(v, ce(v)) = min\{d[u]+c(u, ce(u))|u \in S\}$. Then the final distance from the source to $ce(v)$ is given by $d[v]+c(v, ce(v))$. Also $d[u]$ for u in S are all correct shortest distances from the source.*

Proof If there is a shorter distance to $ce(v)$, it must come from some u in S with $d[u] + c(path(u, ce(v)))$, where $c(path(u, ce(v)))$ is the cost of some path, $path(u, ce(v))$, from u to $ce(v)$ and the first edge on the path goes out of S. From Lemma 3 below, the endpoints of edges from u shorter than $(u, ce(u))$ are all in S, and thus this first edge must be longer than or equal to $(u, ce(u))$. Then this distance must be greater than or equal to $d[v] + c(v, ce(v))$

defined above, a contradiction. Thus, the shortest distance to $ce(v)$ is correctly computed and S is a correct solution set after inclusion of $ce(v)$.

Lemma 3 *For any $v \in S$, vertices in the edge list of v from position 1 to $P[v]-1$ are all in S. Also $c(u, ce(u)) \leq c(u, w)$ for any edge (v, w) such that $w \notin S$.*

Proof From the nature of the algorithm, the pointer movement stops whenever the algorithm finds a candidate outside S. It may stop without finding a candidate outside S.

Setting $S = V$ yields the following theorem:

Theorem 1 *Any algorithm that is a variation of Spira's algorithm with limited scan is correct*

4.5.2 Analysis of the New Algorithm

Lemma 4 *The* find-min *operation at line 10 is executed $O(n)$ times, on average.*

Proof Let p_j be the probability that $v = ce(u_0)$ is clean at line 12 when $|S| = j$. Then $p_j = \frac{n-j}{n-j+N}$. It holds that $p_j = \frac{(n-j)}{n}$, when $j < N$, and $\frac{(n-j)}{(n-j+N)}$, otherwise. Thus, $p_j \geq \frac{n-j}{n-j+N}$ for all j. Since the expected number of trials for $ce(u_0)$ being clean is $\frac{1}{p_j}$, the expected number of *find-min* executions as

$$\begin{aligned}\sum_{j=1}^{n-1} \frac{1}{p_j} &\leq \sum_{j=1}^{n-1} \frac{n-j+N}{n-j} \\ &= \sum_{j=1}^{n-1} \left(1 + \frac{N}{n-j}\right) \\ &= O(n)\end{aligned}$$

As each *find-min* requires $O(1)$ time, the expected time for all *find-min* is $O(n)$. Now *update* is analyzed in two components. One is the time for heap operations, the other being the scanning efforts. For the initial case of $j < N$, v_0 is undefined. Thus in the following summations, j starts from N.

Lemma 5 *The expected number of comparisons in* update *is $O(n \log n)$ in total.*

Proof *Increase-key* or *insert* is performed at the end of each *update*, spending $O(\log n)$ time. The *update* at line 15 is done $n-1$ times, meaning $O(n \log n)$ time for this part. The u_0 given at line 10, is updated at line 20. This part takes $O(n \log n)$ time as line 10 is executed in $O(n)$ time on average. The analysis on general u in line 16-18 follows. Since u is already in

Q, *increase-key* will take place. The batch processing is done for all *increase-key*'s at line 31 for each $L[v_0]$. That is, after all changes of key values are done for all $u \in L[v_0]$. The tree is organized back to a heap in the bottom-up fashion in the same way as in (1). The probability that $ce(u) = v_0$ for u is $\frac{1}{(n-j+N)}$. Interpreting this probability as p and the size of the heap as j in Lemma 1, the time for the restoration of the heap is bounded by the following summation.

$$\begin{aligned}\sum_{j=N}^{n-1} \left(\frac{j}{n-j+N} + \log j \right) &\leq \sum_{j=1}^{n-1} \left(\frac{j}{n-j+N} + \log j \right) \\ &= O(n \log n)\end{aligned}$$

Thus the expected total time for comparisons in *update* is $O(n \log n)$.

Lemma 6 *The total scanning effort is* $O(n \log n)$.

Proof The scanning effort of *update(v)* and *update(u_0)* at lines 15 and 20 is $O(n \log n)$ in total since these lines are executed $O(n)$ times and each takes $O(\log n)$ time. The probability that $u \in S$ has v_0 as its candidate is $\frac{1}{(n-j+N)}$. There are j such members in S, resulting in the expected number of such u's being $\frac{j}{(n-j+N)}$. The probability that the candidate hitting the valid area is $\frac{(n-j+N)}{n}$. Thus the scanning effort for each such u is $\frac{n}{(n-j+N)}$. From endpoint independence, those two values can be multiplied and the expected scanning effort for all u_0 at line 18 is

$$\begin{aligned}\sum_{j=N}^{n-1} \frac{jn}{(n-j+N)^2} &\leq \sum_{j=1}^{n-1} \frac{jn}{(n-j+N)^2} \\ &\leq \sum_{j=1}^{n-1} \frac{n^2}{(n-j+N)^2} \\ &= \sum_{j=1}^{n-1} \frac{1}{(1 - \frac{j}{n} + \frac{1}{\log n})^2} \\ &= \sum_{j=1}^{n-1} \frac{n}{(1 - \frac{j}{n} + \frac{1}{\log n})^2} \times \frac{1}{n} \\ &= \int_0^1 \frac{n}{(1 - x + \frac{1}{\log n})^2} \mathrm{dx} \\ &= O(n \log n)\end{aligned}$$

From those lemmas, the following theorem can be reached.

Theorem 2 *The expected running time of Algorithm 9 is $O(n \log n)$, and its APSP version runs in $O(n^2 \log n)$ time on average.*

4.6 Algorithm Implementation Details

The algorithms presented in this thesis use complete dense directed graphs. This means, $m = n(n-1)$, where n is the number of vertices and m is the number of edges. Edge costs are randomly generated with non-negative edge costs with no self-loop.

All algorithm implementations were written in the C programming language. The same programming style was used to program all algorithms. These programs were compiled using the gcc compiler. To get the all pairs shortest path (APSP) results, n single source shortest path(SSSP) problem was solved.

The maximum problem graph size, a graph with n vertices and m edges, used in this experiments was limited by available RAM. To measure run time, varies samples of graphs were generated due to variation in edge costs which occurred among randomly generated graphs.

The results on page 96 to 98 were obtained from the experiments that had been done using Intel(R) Xeon (R) CPU E5645 @ 2.40Ghz, 4.0 Gb of RAM machine, running on Ubuntu Linux operating system, at Sultan Idris Education University, Malaysia. In these experiments, for the number of vertices, $n \leq 1500$, 50 samples of graphs were used. For $n = 2000$ and $n = 2500$, 20 and 10 samples of graphs were used. The variety of the number of samples chosen was due to the performance of the hardware used to run the program.

The results showed on page 99 - 103 were obtained using an Intel(R) Core(TM) 2 Quad CPU Q8400 @ 2.66Ghz, 3.24 Gb RAM machine running the Fedora Linux operating system, at the University of Canterbury, New Zealand. In these experiments, 10 graph samples were used, from $n = 500$ to $n = 2500$.

Algorithms compared here sort the edges from each vertex v in non-decreasing order of edge costs beforehand, in a method called *pre-sort.* These algorithms use a binary heap to match them with main operations such as *find-min*, *increase-key* and *insert* operations. For pre-sorting, a quicksort technique was used. Efficiency and a fast sorting method were the reasons why this sorting technique was chosen.

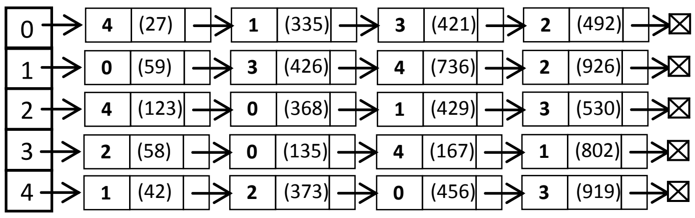
In this thesis, the efficiency of algorithms is compared mainly by calculating the number of key comparisons in the heap operations and in the algorithms themselves. This measurement is chosen because it is not only machine independent, but also the most expensive operation in these algorithms.

To see whether the algorithms that have been developed are correct, two methods can be used. The first method is to compare the result obtained from the experiment with results produced by Floyd's algorithm. The shortest distance results, generated from Floyd's algorithm are guaranteed to be correct as this algorithm is implemented as a simple tight product of three nested loops as explained in chapter 2. If the shortest path results obtained from an algorithm are equal to the results obtained by Floyd's algorithm, the algorithm is said to be correct. The experiment suggests that the algorithms were accurate, as the shortest distance results obtained were equivalent to those produced by Floyd.

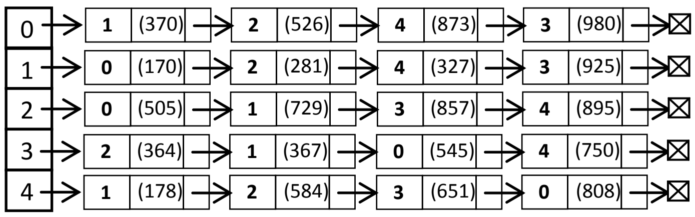
The second method which can be used is to get the results using hand calculation that can also be called manual calculation. This can be done when the size of graph is very small as it is easier to trace the result step by step. In this method, an assumption can be made; if the total number of key comparisons generated by the program are equal to the total number of key comparisons using hand calculation, the algorithm can be accepted as correct. An example of the second method used to test the correctness of MT and the new algorithm is further decribed.

In the experiment that was carried out, three samples of graphs had been generated for a single n vertices. The three graph samples that have been used are shown in Figure 4.8. A manual counter was used to calculate the total number of key comparisons at three places: in the heap operations, in the algorithm when the $d[v]$'s key were compared, and in the algorithm when scanning was being done to choose the best candidates. When ever a new key comparison was needed, the counter value would incrementally increase by one. The results obtained using the manual calculation method was then be compared with the results obtained when the algorithms were run using a compiler. The experiment results showed that our algorithms were correct, as the results obtained from the program execution were equivalent to the results obtained using manual calculation.

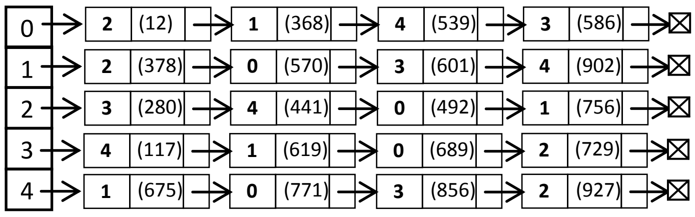
To understand the results better, mean and standard deviation for each sample were calculated. To do this, two functions had been included in the test program to calculate the mean and standard deviation. Table 4.1 shows the results obtained from this experiment. With widely different means, the coefficient of variation(CV) was used to interpret the results instead of the standard deviation. From the results obtained as shown in Table 4.1, CVs for the both algorithms are less than 10%. This suggests that good results have been obtained, as the range



(a) The first graph sample



(b) The second graph sample



(c) The third graph sample

Figure 4.8: Three different graphs are generated with random edge costs for $n = 5$. The graphs are represented using adjacency lists

of the total number of key comparisons are close to the total number of key comparisons of the mean.

4.7 Experimental Results and Analysis

This section presents the results of the experimental comparisons of algorithms.

The main purpose of these experiments was to compare the new algorithm with the Moffat-Takaoka(MT) algorithm. MT algorithm defines a *limit* variable to distinguish between the two phases. The revised version of MT algorithm as shown in Algorithm 8 was used rather than the previous version of MT algorithm as shown in Algorithm 7 for its close similarity to the new algorithm.

Algorithm	**G1**	**G2**	**G3**	**Min**	**Max**	**Mean**	**SD**	**CV**
MT	155	156	136	136	156	149.00	9.20	6.2%
The New Algorithm	163	176	138	138	176	159.00	15.77	9.9%

Table 4.1: The total number of key comparison for MT and the new algorithm when three samples of graphs, $n = 5$ in Figure 4.8 are used. G1, G2 and G3 represent the graphs used in the experiment.

Results. Our experimental results show that the results obtained between these two algorithms are quite close. MT algorithm shows slightly better performance than the new algorithm. However, the new deterministic algorithm for the APSP is provided as an alternative to the existing MT algorithm. The major advantages of this approrach compared to the MT algorithm are its simplicity, intuitive appeal and ease of analysis. Moreover, the algorithm is shown to be reliable as the expected running time is $O(n^2 \log n)$. When comparing results of the APSP obtained from this algorithm with Floyd's algorithm, the same results are achieved; it means the algorithm is correct with respect to a specification. To our knowledge, this is the first alternative algorithm that solves the APSP in $O(n^2 \log n)$ expected time. For almost 35 years, MT algorithm remains the only fast algorithm within the context. The use of a critical point to divide the algorithm into two phases makes the algorithm hard to analyze. The probabilistic dependence before and after the critical point produces over-run that might be overlooked. A conceptual contribution of this algorithm is not only it can solve the APSP problem in $O(n^2 \log n)$ expected time, but also make the analysis better by removing the over-run analysis. The details of the results follow:
The results to confirm that the new algorithm is run in $O(n^2 \log n)$ expected time as it has been proved in the previous section are given first. To do this, the total number of key comparisons of the new algorithm obtained from the experiment, is divided by $(n^2 \log n)$. The near-constant values obtained as shown in Table 4.2, confirm that the running time of this new algorithm is $O(n^2 \log n)$ expected time. Note that the units shown in all tables are number of key comparisons.

Input Size, n	**New Algo** $(\times 10^7)$	$n^2 \log n (\times 10^7)$	$\frac{\textbf{New Algo}}{n^2 \log n}$
500	1.14	0.22	5.09
1000	5.02	1.00	5.03
1500	11.56	2.37	4.87
2000	20.14	4.39	4.59
2500	32.57	7.05	4.62

Table 4.2: The total number of key comparison for the new algorithm

Table 4.3 shows the number of key comparisons when MT algorithm has been run. The minimum and the maximum number of key comparisons between the samples have been recorded when executing the program.

Input Size, n	**Min** ($\times 10^7$)	**Max** ($\times 10^7$)	**Mean** ($\times 10^7$)	**SD** ($\times 10^7$)	**CV** (%)
500	0.87	1.18	1.01	0.07	7.19
1000	3.98	5.43	4.51	0.29	6.54
1500	9.27	13.09	10.51	0.98	9.29
2000	16.57	20.93	18.42	1.34	7.28
2500	26.65	38.15	30.04	2.89	9.62

Table 4.3: The total number of key comparison for MT algorithm

For the new algorithms, the results is shown in Table 4.4.

Input Size, n	**Min** ($\times 10^7$)	**Max** ($\times 10^7$)	**Mean** ($\times 10^7$)	**SD** ($\times 10^7$)	**CV**(%)
500	0.99	1.32	1.14	0.07	6.69
1000	4.48	5.94	5.01	0.29	5.91
1500	10.31	14.07	11.55	0.97	8.41
2000	18.20	22.71	20.14	1.35	6.73
2500	29.10	40.78	32.57	2.94	9.03

Table 4.4: The total number of key comparison for the new algorithm

From the results obtained in Table 4.3 and 4.4, it is suggested that the both algorithms are correct. For example, in the new algorithm, the target set to which the end point of candidate is sought is changing all the time, which causes more randomness. Therefore the standard deviation is quite large. Comparing the CV of both the algorithms, it can be suggested that the dispersion of the number of key comparisons over the mean are equal for the both algorithms as they are less than 10%.

To see which algorithm shows better performance, the total number of key comparisons of MT algorithm was compared with the total number of key comparisons of the new algorithm. The result is shown in Table 4.5. The mean of both algorithms is used to represent the total number of key comparisons.

Input Size, n	**MT** ($\times 10^7$)	**The New Algorithm** ($\times 10^7$)
500	1.01	1.14
1000	4.51	5.01
1500	10.51	11.55
2000	18.42	20.14
2500	30.04	32.57

Table 4.5: The total number of key comparison between MT and the new algorithm

It is interesting to see how many samples of graphs are needed to run for a specific n vertices. Is 10 samples of graphs enough to test these algorithms? Table 4.6 shows the total number of key comparison when only 10 samples of graphs have been used for different n vertices. Means, standard deviations and coefficient of variations were calculated by the program when the both algorithm were being executed. Table 4.7 summarizes the findings.

Input Size, n	**Minimum** $\times 10^7$		**Maximum** $\times 10^7$		**Mean** $\times 10^7$		**SD** $\times 10^7$		**CV** %	
	MT	New	MT	New	MT	New	MT	New	MT	New
500	0.93	1.06	1.12	1.27	1.00	1.13	0.06	0.06	5.7	5.4
1000	3.93	4.40	5.90	6.41	4.47	4.98	0.57	0.58	12.6	11.6
1500	9.29	10.28	11.32	12.31	10.19	11.22	0.75	1	0.73	6.4
2000	17.18	18.74	21.00	22.69	18.82	20.52	1.12	1.14	5.9	5.5
2500	26.88	29.38	33.64	36.19	29.67	32.18	1.88	1.89	6.3	5.9

Table 4.6: The total number of key comparison between MT and the new algorithm. The "New" represents the new algorithm

Input Size, n	**MT** ($\times 10^7$)	**The New Algorithm** ($\times 10^7$)
500	1.00	1.13
1000	4.47	4.98
1500	10.18	11.22
2000	18.81	20.52
2500	29.67	32.17

Table 4.7: The total number of key comparison between MT and the new algorithm when a graph sample, $s = 10$

It is hard to say whether 10 samples of graphs are enough to test these algorithm as when $n = 1000$, CV $> 10\%$. However, the findings shows that better CV is achieved with other n vertices, CV $\leq 10\%$.

Before an assumption could be made for the distribution of the number of key comparisons, the two algorithms were then run 100 times each with $n = 1000$, and the mean, SD and CV were recorded. The results are shown is Table 4.8.

When $n = 1000$ and samples, $s = 10$, CV $> 10\%$. However, when the same number of vertices , $n = 1000$ uses 100 samples of graphs, CV $\leq 10\%$. From the results it can be suggested that when the number of samples are very large, better CV will be obtained. An assumption has been made by the results obtained in Table 4.6 that 10 samples of graphs are enough to compare the MT and the new algorithm proposed in this thesis. This is mainly

Items	**MT**	**The New Algorithm**
Min ($\times 10^7$)	3.92	4.41
Max ($\times 10^7$)	5.77	6.28
Mean ($\times 10^7$)	4.45	4.95
SD ($\times 10^7$)	0.36	0.36
CV	8.1%	7.3%

Table 4.8: The total number of key comparison between MT and the new algorithm with $n = 1000$

because of from $n = 500$ to $n = 2500$, 80% of CVs are less than 10% even though only 10 samples of graphs have been used in the experiments.

To give better results, experiments to compare the performance between MT and the new algorithm have been done on a different machine. When a different machine had been used, the results obtained are shown in Table 4.9.

Input Size, n	**MT** ($\times 10^7$)	**The New Algorithm** ($\times 10^7$)
500	1.00	1.13
1000	4.47	4.98
1500	10.18	11.22
2000	18.81	20.52
2500	29.67	32.17

Table 4.9: The total number of key comparison between MT and the new algorithm using a different machine

Even though different machines with different specifications and locations were used, the results obtained from the both machines were similar.

If the CPU computation time is concerned, the running time obtained for both algorithm when CPU time is calculated is shown in Figure 4.9.

Input Size, n	**MT (seconds)**	**The New Algorithm (seconds)**
500	0.35	0.42
1000	1.62	1.79
1500	3.92	4.22
2000	7.20	7.95
2500	11.77	12.95

Table 4.10: Running times for the APSP algorithms

The CPU running time of the new algorithm does not perform as good as the MT algorithm. This is because of the implementation of the new algorithm requires the use of two-dimensional arrays more often than the MT algorithm. Using two-dimensional arrays is relatively slow in the C language.

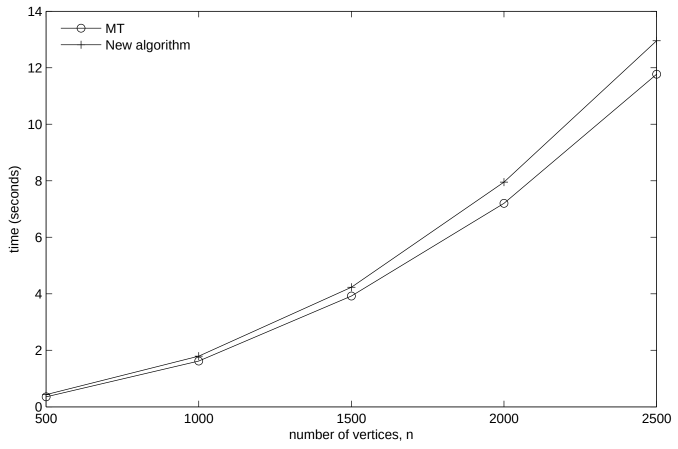


Figure 4.9: Running times for the APSP algorithms

The results shown in Table 4.5 and graph in Figure 4.9 demonstrate that the MT algorithm is a bit superior than the new algorithm. To see the reasons why the number of comparisons in the new algorithm had been slighly higher, further experiments were done by dividing the complexity of key comparisons into a few different parts.

In general, there are three places where the key comparison is done: in heap operations such as when executing *insert*, *find-min* and *increase-key* operations, in the algorithm when the minimum $d[v]$ is chosen and in the algorithm when scanning process to find the candidate is performed. Table 4.11 lists the results of the both algorithms if only key comparisons in heaps operations are considered.

Input Size, n	**MT** $(\times 10^7)$	**The New Algorithm** $(\times 10^7)$
500	0.55	0.67
1000	2.44	2.90
1500	5.41	6.33
2000	9.96	11.45
2500	15.08	17.28

Table 4.11: The total number of key comparison in heap operations

The results show that between 50 to 60% of the total complexity are derived from this operation. The both algorithms require almost the equal number of key comparison from the total complexity.

Key comparison is also needed when comparing the minimum $d[v]$ value in the *update* procedure. If this is the only measurement used as key comparisons, results obtained are shown in Table 4.12. The number of key comparisons in this part consumes less than 10% from the total complexity. Both algorithms show almost the same results.

Input Size, n	**MT** $(\times 10^7)$	**The New Algorithm** $(\times 10^7)$
500	0.09	0.11
1000	0.38	0.43
1500	0.86	0.98
2000	1.56	1.78
2500	2.46	2.78

Table 4.12: The total number of key comparison in comparing the $d[v]$'s values

Scanning to get the candidate for each vertex v of the both algorithms also requires key comparisons to be measured. The scanning technique used to get the next candidate varies between these two algorithms, as explained in the earlier sections. Table 4.13 shows the results when scanning is chosen as key comparisons.

Input Size, n	**MT** $(\times 10^7)$	**The New Algorithm** $(\times 10^7)$
500	0.36	0.36
1000	1.65	1.65
1500	3.91	3.91
2000	7.30	7.28
2500	12.13	12.11

Table 4.13: The total number of key comparison in scanning part

In Table 4.13, the new algorithm requires less or almost the same pointer movement such in the MT algorithm. Both algorithms spend between 35 to 40% comparing keys from the total complexity. Even though the new algorithm does not require the changes of behaviour at certain point, the results obtained show that a clever method is deployed in this new algorithm. In other words, the new algorithm always tries to balance this scanning technique with the overall complexity. This is a novel method introduced by the new algorithm.

Results obtained in Table 4.11, 4.12 and Table 4.13 describe that when the number of scanning to get the best candidate is increased, then the number of key comparisons to perform heap operations is decreased. This situation leads to a situation similar to Dantzig's algorithm. The MT algorithm defines the critical point to avoid this scenario. When the size of the solution set is close to n, more scanning is done, which requires more key comparisons. The new algorithm removes the critical point by trying to balance the scanning effort when expanding the solution set. However, the effort is improved, the total performance still can not beat the MT alfgorithm. To balance between key comparisons in many parts, the new algorithm uses fewer key comparisons in scanning, resulting in more key comparisons used in the heap operations. This is the main reason why the new algorithm achieves slightly lower performance than the MT algorithm when the number of key comparisons is measured.

The buffer zone, N of size $\frac{n}{\log n}$ in the new algorithm can be parameterized as kN, depending on the characteristic of the computer used with specific speeds for comparisons and pointer movements. Results of this tuning are shown in Table 4.14. Units are 10^7.

		New Algorithm (varies k)				
n	**MT algorithm**	**0.6**	**0.8**	**1.0**	**1.2**	**1.4**
500	1.00	1.14	1.13	1.13	1.15	1.18
1000	4.47	5.04	4.98	4.98	5.04	5.14
1500	10.19	11.40	11.21	11.22	11.25	11.39
2000	18.82	20.94	20.51	20.52	20.63	20.93

Table 4.14: The total number of key comparison with different k parameters

These results demonstrate that when the k value is changed, the total number of key comparison will also be slightly changed. The best k parameter to use in this new algorithm is when the value of $k = 0.8$, as shown in Table 4.14.

Another finding from these experiments regards the performance of key comparison when some modification is made to the *inrease-key* operation in the heap. This modification is required to maintain a batch list from the algorithms. The *increase-key* function is called to increase the key value of a node. This will be done by the priority queue. When there are two keys in the algorithm that need to be updated, then, the *increase-key* function will be called twice to update each keys. The more the function is executed, the more the key comparisons is done. In the algorithms that have been discussed here, when the edge list of $L[v]$ has to be updated, that means there might be more than one keys have to be revised. This therefore increases the total number of key comparisons. To avoid this, updating those $u \in L[v]$ should be done in a batch list; the $L[v]$ is sent to the *increase-key* function in a batch and the heap will change the keys with the new keys and re-balance the heap after all keys have been updated. Increasing key values in a batch can reduce the number of key comparisons. Table 4.15 demonstrates the total number of key comparisons which can be recovered when the batch processing technique is used.

Results in Table 4.15 were obtained by deducting the total number of key comparisons received by a single *inrease-key* operation with a batch processing *increase-key*. The results

Input Size, n	**MT**	**The New Algorithm**
500	6829.50	1923.20
1000	14665.40	4148.40
1500	17890.70	6509.40
2000	22554.80	7940.20
2500	30633.10	10497.80

Table 4.15: The total number of key comparison that can be saved if a batch processing technique is used in *increase-key* function

show that the MT algorithm gains three times more benefit from this technique than the new algorithm. Batch processing is done when all $u \in L[v]$, need to be updated to find other candidates. The MT algorithm requires this update before the critical point, however it relaxs the idea after the critical point. The new algorithm, on the other hand, requires this update for all $u \in S$ that are pointing to the expiring vertex, v_0. The batch list is called $L[v_0]$. The total number of $u \in L[v]$ is more than $u \in L[v_0]$. Therefore, the total complexity in the MT algorithm is reduced by this technique. This is one of the other factors contributing to the better performance of the MT algorithm.

From the experiment and the analysis that have been done, it is suggested that the new algorithm is simpler than the MT algorithm in three aspects: analysis, design and code implementation.

Simpler in analysis: In the second phase of MT algorithm, each candidate of the vertices in the solution set S is located in U. The set U is the set of unlabeled vertices, where at the end of the first phase, the size of U, $|U|$ is $\lfloor \frac{n}{\log n} \rfloor$ by design. As the candidates of every vertices in S must be in U at the end of Phase 1, Moffat and Takaoka assume that the vertices in U are randomly scattered throughout each adjacency list. It is expected that $\log n$ of edges are required to be scanned. Thus, the total scanning effort for the first phase is $O(n \log n)$ as $n \log n$ edges will be scanned. However, Mehlhorn and Priebe (50) argue this assumption is inaccurate as they point out that U is determined by the orderings of the adjacency lists and cannot be fixed independently. They state that the total of $n - \lfloor \frac{n}{\log n} \rfloor$ edges out of the source are scanned in the first phase. The last $\lfloor \frac{n}{\log n} \rfloor$ edges in the adjacency list of the source will determine the vertices in U. This can be obtained by assuming that the edge costs for all outgoing edges from the source are 1, and the rest are 2. In (50), the authors also argue whether the number of iterations needed to expand the size of S in Phase 2 and the scanning effort needed in the update procedure are independent or not. The changing process from the first phase to the second phase in the MT algorithm makes the algorithm very difficult to analyze. The new algorithm, however, is easier to analyze as there is no changing process from the beginning to the end when solving a problem.

Simpler in design: In each iteration process for the expansion of the solution set S, there are certain steps to be followed. At first S is empty. The first vertex to be included in S is a source vertex s. The size of S is expanded when a new vertex is inserted in S. In the MT algorithm, when the number of vertices in the solution set $|S| = n - \frac{n}{\log n}$, some modifications to the steps in the iteration have to be made. This is mainly because the critical point has been reached and the algorithm is entering the second phase. During the second phase, the iteration should be modified to perform other procedures as the candidates of every vertices in S must be vertices in the U set only. On the other hand, the new proposed algorithm promotes the same way of solving the problem from the beginning, when S is empty until all vertices have been included in S. The new algorithm has an even consistency or follows the same steps all the way to solve a particular problem. The same heap data structure is also used all the way. That is why the new algorithm is simpler compared to the MT algorithm.

Simpler in code implementation: As the iteration to include the vertices in the solution set is the same for all vertices, the code is simpler in implementation as no critical point is needed to be checked. Our experiments show that the line of codes to write the algorithm is slightly higher in the MT algorithm compared to the new algorithm even though the same programming style for the inclusion of vertices in the solution set is used.

In conclusion, the design of the new algorithm is very promising. It balances heap operations, which are measured by the number of node-to-node key comparison, and scanning effort, which is measured by pointer movements. In other words, the new algorithm balances two different complexities of different natures. It also permits a simple and easy implementation in one of the most crtical steps in the existing MT algorithm, yet the new algorithm is easier to analyze. Even though the current performance of the new algorithm is not strong, as technology improves and computers grow more powerful, it will be possible to add more features to improve the performance of the new algorithm.

5

Concluding Remarks and Future Work

The primary objective of this thesis is to solve the all pairs shortest paths problem (APSP) most efficiently. This was achieved through the development of a new shortest path algorithm which attained optimal complexity of $O(n^2 \log n)$ with a simpler analysis. The analysis is done based on the average case analysis. The existing well known algorithm, Moffat-Takaoka (MT) algorithm which solves the APSP in the same time complexity employed two phases in solving the problem. The introduction of a critical point to divide the two phases makes the algorithm superior and unbeatable. No matter how much, the critical point has caused a tougher analysis, yet, it is still difficult to understand the algorithm itself. However, the new algorithm introduced in this thesis removes the critical point concept.

The MT algorithm was developed in 1985. Since then, no other algorithm can solve the problem better than that for a dense non-negative digraph. When this MT algorithm was presented, the analysis of this algorithm had been reviewed by other authors as it had flaws in the algorithm analysis. It appeared that the probabilistic dependence before and after the critical point produces over-run have been overlooked. On the other hand, it has been shown that this algorithm is relatively faster and it manages to obtain the best time complexity.

What motivates the introduction of a new algorithm was to challenge MT algorithm's time complexity which, the research also aimed to produce an alternative algorithm that can be used to solve the APSP in the same way. To achieve this, the research found two elements contributing to the performance of the existing algorithm; scanning method used to select the best nodes for the operation and the frequency of the heap operations called by the algorithm.

This research has successfully removed the critical point concept by balancing these two complexities from the beginning until the end. In other words, the algorithm runs smoothly and steadily without having to change its behavior along the process, having the the same scanning technique being used through out the runtime. Even though the experimental results did not achieve the expected shown best performances, somehow providing the alternative algorithm is a great achievement. A conceptual contribution of this algorithm does not only solves the APSP problem in $O(n^2 \log n)$ expected time, but also improves by removing the over-run analysis.

Guided by the importance of using a good data structure to speed up the process of solving the shortest path problem, this thesis too explored the priority queues or heaps areas. This is in fact the second contribution of this thesis. This research presented two new data structures called quaternary and dimensional heaps. This research has also shown that under certain circumstances, it is possible for the new data structures to outperform the existing well known heaps such as trinomial, binary, Fibonacci and 2-3 heaps.

The first data structure known as a quaternary heap is very comparable to the trinomial heap. This data structure was developed to see the performance of a dense structure heap with special input sequences. Thus, an insertion cache was used in this heap. Overall performance obtained from the experimental results showed that this heap could beat the trinomial heap when the number of vertices, n was relatively small. The research then explored the performance of a thin structure heap with the use of workspace and dimension when performing a decrease-key operation in the heap. The objective was to produce a better decrease-key function tohelp speed up the update key process. Experimental results showed that when m times decrease-key operations were called, where m denoted the number of edges in a graph, the dimensional heap gave outstanding results and could easily beat other heaps performances such as binary, fibonacci and 2-3 heaps

5.1 Measurement Technique

In this thesis, the efficiency of algorithms and data structures was measured by calculating the number of key comparisons. A single key comparison was done when there were two key values of two nodes to be compared before executing certain operations. Actual CPU time computation was not used as the measurement. This was mainly because of the concept of computer itself. It is well known that every computer performs different operations at different speeds, varying in memory size, processor and disk speeds.

The measurement in the experiment was done more abstractly by counting the number of basic operations required to run the algorithm. The key comparison was chosen since it was the basic operation and the most expensive operation for this algorithm. It is also machine independent.

5.2 Future Research

By reflecting on the results obtained from this research, several interesting research topics and recommendations have been observed. These would allow other researchers to further evaluate and expand the findings of this thesis.

5.2.1 Better scanning technique

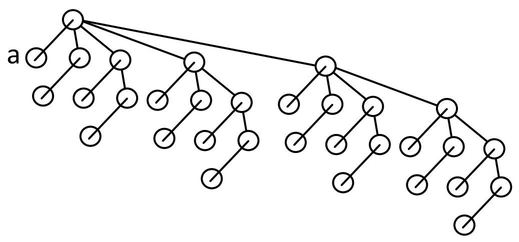
The difference between one algorithm and another which solves the all pairs shortest path problem (APSP) in different time complexities, is the scanning technique used to find a suitable node to be explored after the insertion of the adjacent node. This scanning process is done to expand the size of the solution set and this process is the most expensive operation in the algorithm. The equally expensive operation is the key comparisons needed by a heap data structure. However, these two expensive operations are related to each other. The more scanning operation is done, the less key comparisons used by the data structure and it is the same the other way around. If these two complexities are balanced, a faster algorithm to solve the APSP is possible. Can we perhaps create a perfect balance between these two complexities?

5.2.2 Using a pair-wise algorithm

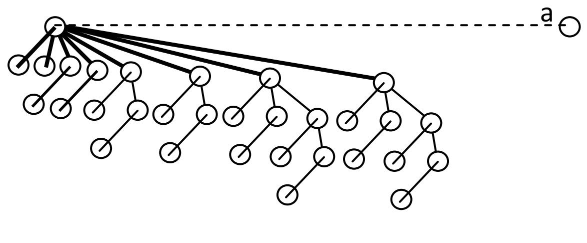
The existing best algorithms usually solve the APSP algorithm by performing $n\times$ single source shortest path (SSSP) problems. If a pair-wise algorithm (53) is used to solve the problem, can we get better performance? The idea of the pair-wise algorithm comes from the hidden path algorithm (54). To solve the all pairs shortest path problem then , Dijkstra's algorithm is executed in parallel to all nodes in the graph. Using limited edge costs, (55) shows that the APSP problem can be solved in $O(mn + n^2 \log(\frac{c}{n}))$, where c is the maximum edge cost, n is the number of vertices and m is the number of edges in a graph. The algorithm in (55) uses a pair-wise algorithm with a cascading bucket system data structure. Research using this technique could be explored and the best data structure to use with this algorithm also should be developed. It is an open problem of which technique is the best to use.

5.2.3 The best constant time for the decrease-key operation

A potential value for each edge is assigned using the amortized cost analysis. In the dimensional heap, each thin edge is assigned to have potential 1, $\Phi = 1$ while two thick edges are assigned to have potential 1 for the each edge, thus $\Phi = 2$ for two thick edges. When the heap structure is very thin, performing a decrease key operation results in many thick edges. Figure 5.1 shows the decrease-key operation on node a. Figure 5.1(a) describes the heap structure before the operation and Figure 5.1(b) shows the structure after the decrease-key operation has been performed.



(a) The heap structure before the decrease-key



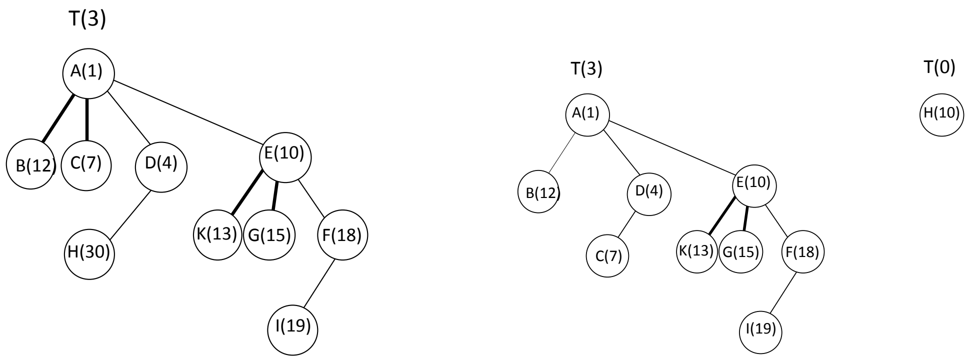
(b) The heap structure after the decrease-key

Figure 5.1: A decrease-key operation is called to decrease key value of node a

When the thinnest structure becomes the dense structure after the decrease-key operation, the potential is reduced at least by one. For example, the potential for the tree structure in 5.1(a) is 31, $\Phi = 31$ and the potential of 5.1(b) is 30, $\Phi = 30$. When $key(a)$ is decreased, no key comparison is made. The amortized cost for this operation is given by:

$$a_i = t_i - (\Phi_i - \Phi_{i-1}) = 0 - (30 - 31) = 1 \tag{5.1}$$

When the decrease-key operation is performed, the tree must be restructured. The restructuring propagates to higher dimensions. With the thinnest structure, going to the higher dimension is not that expensive. This is the merit of the dimensional heap; it provides simplicity for the work space and is generally more efficient in the decrease-key operation than other heaps.



(a) Before $key(H)$ is decreased (b) The result after $key(H)$ was decreased

Figure 5.2: Decrease-key operation on node H. The lower dimension sibling of parent's node is thick

However, consider an example of a dimensional tree with a combination of thick and thin edges as shown in Figure 5.2. As there is a need of one key comparison for the decrease-key process, then

$$a_i = t_i - (\Phi_i - \Phi_{i-1}) = 1 - (8 - 9) = 2 \tag{5.2}$$

The question is, can we reduce the constant factor obtained by 5.2? Is it possible to change the heap structure of this type of tree after the decrease-key operation with other heap structure such as 2-3 heap? The idea is to make the constant factor from the amortized cost as low as possible. This is mainly because if the running time becomes important, then the constants and low-order terms do matter in terms of which algorithm is actually faster. Research in this area should also be explored.

5.3 Concluding Remarks

Simplicity is one of the most important criteria needed for a good algorithm (56). Algorithmic design should be clearly stated and easily understood by those who are going to convert the algorithm into a real-world program. There is often a conflict in selecting between a fast and a simple algorithm. Typically, the more complex the algorithm, the more difficult it is to implement. Thus, an algorithm might easily encounter subtle bugs after it has been in use for a substantial period of time. Choosing the best algorithm must be done wisely.

With future advances in technology such as CPUs, there is always a chance that better algorithms can be created as well as data structures that can work perfectly with the algorithm. CPU hardware and features are rapidly evolving. We should not be too concerned about improving the efficiency of an algorithm, since what is considered inefficient today will in turn become at least adequately efficient on the faster hardware within a few years' time.

References

[1] Alistair Moffat and Tadao Takaoka. **An all pairs shortest path algorithm with expected time $O(n^2 \log n)$**. *SIAM J. Comput.*, **16**(6):1023–1031, December 1987. 6, 21, 82, 85, 86, 87, 92

[2] Tadao Takaoka. **Theory of Trinomial Heaps**. In *Proceedings of the 6th Annual International Conference on Computing and Combinatorics*, COCOON '00, pages 362–372, London, UK, UK, 2000. Springer-Verlag. 6, 19, 23, 30, 34, 35, 36, 38, 47

[3] Michael L. Fredman and Robert Endre Tarjan. **Fibonacci heaps and their uses in improved network optimization algorithms**. *J. ACM*, **34**(3):596–615, July 1987. 6, 19, 22, 38, 66

[4] Tadao Takaoka. **Theory of 2-3 heaps**. *Discrete Appl. Math.*, **126**(1):115–128, March 2003. 6, 19, 23, 27, 34, 36, 38, 47, 66, 67

[5] Seth Pettie and Vijaya Ramachandran. **Computing Undirected Shortest Paths with Comparisons and Additions**, 2001. 9

[6] Edsger. W. Dijkstra. **A Note on Two Problems in Connexion with Graphs.** *Numerische Mathematik*, **1**:269–271, 1959. 14, 72

[7] Robert B. Dial. **Algorithm 360: shortest-path forest with topological ordering [H]**. *Commun. ACM*, **12**(11):632–633, November 1969. 14

[8] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. 14, 22

[9] Boris Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. **Shortest Paths Algorithms: Theory And Experimental Evaluation**. *Mathematical Programming*, **73**:129–174, 1993. 14, 16

[10] Mikkel Thorup. **Integer priority queues with decrease key in constant time and the single source shortest paths problem**. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 149–158, New York, NY, USA, 2003. ACM. 14, 22

[11] James B. Orlin, Kamesh Madduri, K. Subramani, and M. Williamson. **A faster algorithm for the single source shortest path problem with few distinct positive lengths**. *J. of Discrete Algorithms*, **8**(2):189–198, June 2010. 15

[12] P.E. Hart, N.J. Nilsson, and B. Raphael. **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**. *Systems Science and Cybernetics, IEEE Transactions on*, **4**(2):100–107, 1968. 15

[13] Ira Pohl. **Bi-directional Search**. *Machine Intelligence*, **6**:127–140, 1971. 15

[14] T. Ikeda, Min-Yao Hsu, Hiroshi Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. **A fast algorithm for finding better routes by AI search techniques**. In *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pages 291–296, 1994. 15

[15] Jrgen Eckerle. **An optimal bidirectional search algorithm**. In Bernhard Nebel and Leonie Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence*, **861** of *Lecture Notes in Computer Science*, pages 394–394. Springer Berlin Heidelberg, 1994. 15

[16] Andrew V. Goldberg and Chris Harrelson. **Computing the Shortest Path: A* Search Meets Graph Theory**. pages 156–165, 2004. 15

[17] Kazuaki Yamaguchi and Sumio Masuda. **An A* Algorithm with a New Heuristic Distance Function for the 2-Terminal Shortest Path Problem**. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, **E89-A**(2):544–550, February 2006. 15

[18] L. Fu, D. Sun, and L. R. Rilett. **Heuristic shortest path algorithms for transportation applications: state of the art**. *Comput. Oper. Res.*, **33**(11):3324–3343, November 2006. 15

[19] Peter Sanders and Dominik Schultes. **Highway Hierarchies Hasten Exact Shortest Path Queries**. In GerthStlting Brodal and Stefano Leonardi, editors, *Algorithms ESA 2005*, **3669** of *Lecture Notes in Computer Science*, pages 568–579. Springer Berlin Heidelberg, 2005. 15

[20] Peter Sanders and Dominik Schultes. **Engineering highway hierarchies**. *J. Exp. Algorithmics*, **17**:1.6:1.1–1.6:1.40, September 2012. 15

[21] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. **Highway Hierarchies Star**. Technical report, ARRIVAL Project, November 2006. work presented at 9th DIMACS Challenge on Shortest Paths. 15

[22] Robert W. Floyd. **Algorithm 97: Shortest path**. *Commun. ACM*, **5**(6):345–, June 1962. 15, 20

[23] Richard Bellman. **On a Routing Problem**. *Quarterly of Applied Mathematics*, **16**:87–90, 1958. 16

[24] Donald B. Johnson. **Efficient Algorithms for Shortest Paths in Sparse Networks**. *J. ACM*, **24**(1):1–13, January 1977. 16

[25] Seth Pettie. **A new approach to all-pairs shortest paths on real-weighted graphs**. *Theor. Comput. Sci.*, **312**(1):47–74, January 2004. 16

[26] Yijie Han and Tadao Takaoka. **An $O(n^3 \frac{\log\log n}{\log^2 n})$ Time Algorithm for All Pairs Shortest Paths**. In *SWAT*, pages 131–141, 2012. 16

[27] Timothy M. Chan. **More algorithms for all-pairs shortest paths in weighted graphs**. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 590–598, New York, NY, USA, 2007. ACM. 16

[28] John William Joseph Wiiliams. **Algorithm 232: Heapsort**. *Communications of the ACM*, **7**(6):596–615, 1964. 19, 22, 66, 72

[29] Philip M. Spira. **A New Algorithm for Finding all Shortest Paths in a Graph of Positive Arcs in Average Time $0(n^2 \log^2 n)$**. *SIAM J. Comput.*, **2**(1):28–32, 1973. 21, 76

[30] Peter Bloniarz. **A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$**. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 378–384, New York, NY, USA, 1980. ACM. 21, 80, 82, 87, 90

[31] Clark Allan Crane. *Linear lists and priority queues as balanced binary trees.* PhD thesis, Stanford, CA, USA, 1972. AAI7220697. 22

[32] Jean Vuillemin. **A data structure for manipulating priority queues**. *Commun. ACM*, **21**(4):309–315, April 1978. 22

[33] Michael L. Fredman. **On the efficiency of pairing heaps and related data structures**. *J. ACM*, **46**(4):473–501, July 1999. 23

[34] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. **Strict fibonacci heaps**. In *Proceedings of the 44th symposium on Theory of Computing*, STOC '12, pages 1177–1184, New York, NY, USA, 2012. ACM. 23

[35] Amr Elmasry, Claus Jensen, and Jyrki Katajainen. **On the power of structural violations in priority queues**. In *Proceedings of the thirteenth Australasian symposium on Theory of computing - Volume 65*, CATS '07, pages 45–53, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. 23

[36] Daniel Dominic Sleator and Robert Endre Tarjan. **Self adjusting heaps**. *SIAM J. Comput.*, **15**(1):52–69, February 1986. 23

[37] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. **Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation**. *Commun. ACM*, **31**(11):1343–1354, November 1988. 23

[38] Seth Pettie. **Towards a Final Analysis of Pairing Heaps**. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, pages 174–183, Washington, DC, USA, 2005. IEEE Computer Society. 23

[39] Amr Elmasry. **Pairing heaps with O(log log n) decrease cost**. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 471–476, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. 23

[40] Haim Kaplan and Robert Endre Tarjan. **Thin heaps, thick heaps**. *ACM Trans. Algorithms*, **4**(1):3:1–3:14, March 2008. 23

[41] Timothy Chan. **Quake heaps: a simple alternative to Fibonacci heaps**. 2009. 23

[42] Siddhartha Sen Bernhard Haeupler and Robert E. Tarjan. **On Rank-Pairing Heaps**. *SIAM J. Comput*, **40**(6):14631485, 2011. 23

[43] Kohei Noshita. **A theorem on the expected complexity of dijkstra's shortest path algorithm**. *Journal of Algorithms*, **6**(3):400 – 408, 1985. 24

[44] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. **Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation**. *Commun. ACM*, **31**(11):1343–1354, November 1988. 36

[45] Andrew V. Goldberg. **A Practical Shortest Path Algorithm with Linear Expected Time**. *SIAM J. Comput.*, **37**(5):1637–1655, February 2008. 66, 67

[46] George B. Dantzig. **On the Shortest Route Through Network**. *Management Science*, **6**(2):187–190, 1960. 73, 79, 81, 83

[47] Tadao Takaoka and Alistair Moffat. **An $O(n^2 \log\log\log n)$ Expected Time Algorithm for the all Shortest Distance Problem**. In *Proceedings of the 9th Symposium on Mathematical Foundations of Computer Science*, MFCS '80, pages 643–655, London, UK, UK, 1980. Springer-Verlag. 80, 81, 82, 87

[48] Tadao Takaoka and Mashitoh Hashim. **A simpler algorithm for the all pairs shortest path problem with o(n2 log n) expected time**. In *Proceedings of the 4th international conference on Combinatorial optimization and applications - Volume Part II*, COCOA'10, pages 195–206, Berlin, Heidelberg, 2010. Springer-Verlag. 83, 87

[49] William Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley and Sons, Inc, Canada, 1971. 85

[50] Kurt Mehlhorn and Volker Priebe. **On the all-pairs shortest-path algorithm of Moffat and Takaoka**. *Random Struct. Algorithms*, **10**(1-2):205–220, February 1997. 85, 86, 87, 103

[51] Andrew V. Goldberg and Robert E. Tarjan. **Expected Performance of Dijkstra's Shortest Path Algorithm**. Technical report, NEC RESEARCH INSTITUTE REPORT, 1996. 87

[52] Mashitoh Hashim and Tadao Takaoka. **A new Algorithm for solving the All pairs shortest path problem in $O(n^2 \log n)$ expected time**. In *in NZCSRSC 2010 conference*, 2010. 87

[53] Tadao Takaoka and Mashitoh Hashim. **Sharing Information in All Pairs Shortest Path Algorithms**. In Alex Potanin and Taso Viglas, editors, *Computing: The Australasian Theory Symposium (CATS 2011)*, **119** of *CRPIT*, pages 131–136, Perth, Australia, 2011. ACS. 107

[54] David R. Karger, Daphne Koller, and Steven J. Phillips. **Finding the hidden path: time bounds for all-pairs shortest paths**. *SIAM Journal on Computing*, **22**(6):1199–1217, 1993. cited By (since 1996) 43. 107

[55] T. Takaoka. **Efficient algorithms for the all pairs shortest path problem with limited edge costs**. **128**, pages 21–26, 2012. cited By (since 1996) 0. 107

[56] Al Aho and Jeff Ullman. *Foundations of Computer Science*. 1992. 110