**PERFORMANCE EVALUATION OF SOME SELECTED VARIANTS OF DIJKSTRA ALGORITHM USING PRIORITY QUEUE**

**IDOWU, Abel Iyanda**

**AAA2000527**

**B.Sc. Ed Computer Science (Ijebu-Ode)**

**A PRE-DATA SUBMITTED TO**

**THE DEPARTMENT OF COMPUTER SCIENCE,FACULTY OFCOMPUTING AND INFORMATICS, LADOKE AKINTOLA UNIVERSITY OF TECHNOLOGY,OGBOMOSO, NIGERIA. IN PARTIALFULFILMENT FOR THE AWARD OF MASTERS OF TECHNOLOGY (M.Tech) IN COMPUTER SCIENCE.**

**FEBRUARY 2024**

**CERTIFICATION**

This pre-data report with the title **performance evaluation of some selected variants of Dijkstra algorithm** submitted by **Idowu, Abel Iyanda** was carried out under our supervision at Ladoke Akintola University of Technology, Ogbomoso, Nigeria.

--------------------------------------------- ---------------------------------

**Supervisor** Date

**Prof. (Mrs.) O. O Alo**, B. Sc, M. Tech. Ph.D.

Professor of computer science

Department of Information System

Ladoke Akintola University of Technology.

Ogbomoso, Oyo State, Nigeria.

---------------------------------------------------- -----------------------------------

**Co-supervisor** Date

**Prof. S. O. Olabiyisi,** B. Tech. M. Tech. M. Sc. Ph.D.

Professor of Computer Science

Department of Computer Science

Ladoke Akintola University of Technology,

Ogbomoso, Oyo State, Nigeria.

**ATTESTATION**

I hereby attest that this research was carried out in the Department of Computer Science, Faculty of Computing and Informatics, Ladoke Akintola University of Technology, Ogbomoso, Nigeria.

----------------------------------------- ------------------------------------

**Head of Department** Date

**Prof. O.O. Adeosun,** B. Sc, M. Tech, Ph.D.

Professor of Computer Science

Department of Computer Science

Ladoke Akintola University of Technology,

Ogbomoso, Oyo State, Nigeria.

**DEDICATION**

This research is dedicated to my creator, the owner of my life.

**ACKNOWLEDGEMENTS**

I give all thanks and adoration to Almighty God, the beginning and the end. The one who was, is and is to come. My profound gratitude goes to my Supervisor Prof. (Mrs.) O. O. Alo and Co-supervisor Prof. S. O. Olabiyisi for their guidance and supervision of the work; I pray that the Lord will increase you in all areas of endeavour in life.

My appreciation also goes to the Head of the Department of Computer Science Prof. O.O Adeosun for his relentless commitment to the academic excellence of postgraduate students in the Department and the entire staff of the Department, my prayer is that you will be favoured, and your heart desires shall be met in Jesus’ name.

I cannot but appreciate the contribution of my dear wife, Mrs. Idowu Abiola Abigeal for her advice and support both in cash and kinds. Similarly my children Victoria, Emmanuel, Gloria, and Daniel, I appreciate you for your understanding. I love you all. God bless you.

Finally, I appreciate my friends, Ogunniyi Olufunmike, Ayoola Olubunmi, Mr., Adeoye and all my fellow postgraduate students in the Department for their support we shall all reach our promise land in Jesus name.

**SUMMARY**

Dijkstra algorithm is an efficient, exact method for finding the shortest paths between vertices in edge and arc weighted graphs. The performance of the Dijkstra algorithm depends solely on the movement update of Dijkstra. The movement update is influenced by computer time, throughput, and computational time complexity. Hence, this research will evaluate the performance of three selected variants (binary heap, binomial heap, and Fibonacci heap) of Dijkstra algorithm using performance metrics by evaluating the performance of each variant of Dijkstra algorithm.

The performance metric influencing the performance of the variants of Dijkstra algorithms will be critically analyze through extensive review of literatures. Experiment will be performed on the three selected variants (binary heap, binomial heap, and Fibonacci heap) of Dijkstra algorithm using Lewis Abstract dataset by evaluating the performance of each of the algorithm using the different population size. The experiment will be conducted in a MATLAB environment. Performance metrics will be used for the evaluation of the experiment. The performance of each algorithm will be evaluated based on throughput, computer time and computational time complexity.

The results to be obtained is expected to show remarkable improvement in the exiting research and serve as benchmark for consequent research.

**TABLE OF CONTENTS**

**Content Page**

Title Page i

Certification ii

Attestation iii

Dedication iv

Acknowledgements v

Summary vi

Table of Content vii

List of Acronyms ix

List of Table x

List of Figure xi

**CHAPTER ONE: INTRODUCTION**

* 1. Background to the Study 1
  2. Statement of the Problem 3
  3. Aim and Objectives 3
  4. Significance of the Study 4
  5. Scope of the Study 4

**CHAPTER TWO: LITERATURE REVIEW**

2.1 Shortest Path Algorithm 5

2.2. Bellman Ford Algorithm 5

2,3. A\* Algorithm 5

2.4 Bi- Directional Dijkstra 6

2.5 Dijkstra Algorithm 7

2.5.1 Simple statement of dijkstra algorithm 9

2.5.2 Dijkstra algorithm procedure 12

2.5.3 Algorithm techniques 13

2.5.4 Techniques of operation of dijkstra algorithm 14

2.6. Variants of Dijkstra Algorithm 17

2.6.1 Binary heap 21

2.6.2 Operation on heaps 21

2.6.3 Application of heaps 24

2.7 Binomial Heap 24

2.7.1 Operation of binomial heap 25

2.7.2 Union operation in binomial heap 26

2.8 Fibonacci Heap 28

2.8.1 Insertion 28

2.8.2 Extract min 28

2.8.3 Consolidation 28

2.8.4 Some interesting facts about Fibonacci heap 31

2.8.5 Advantages of Fibonacci heap 32

2.8.6 Disadvantages of Fibonacci heap 32

2.8.7 Amortized analysis 34

2.9 Performance Metrics Analysis 35

2.9.1 Throughput 35

2.9.2 Computational time complexity 35

2.10Applications of the Shortest Path Algorithm 38

2.11 Importance of Dijkstra Algorithm 39

2.12 Review of Related Works 39

**CHAPTER THREE: METHODOLOGY**

3.1. Overview of Research Approach 45

3.2. Identification of the Performance Metrics 45

3.3. Data Generation from the Experiment 46

3.4 Performance Evaluation of the Experimental Result 47

3.5 Expected Contribution to Knowledge 49

**References**  50

**LIST OF ACRONYMS**

**Lists**

AA A\* Algorithm

ASPP Algorithm for Shortest Path Problems

BDA Bi- directional Dijkstra

BH Binomial Heap

BS Bi-directional search

CT Computer time

DA Dijkstra Algorithm

DSPA Dijkstra Shortest Path Algorithm

FH Fibonacci Heap

GBFS Greedy Best First Search

HO Heap Operation

IMPQ Implementing Moldable Priority Queue

PE Performance Evaluation

SPP Shortest Path Problem

PQ Priority Queue

SPA Shortest Path Algorithm

SPT Shortest Path Tree

SSSPP Single-Source Single-Target Shortest Path Problem

SSPP Single –Source Shortest Path Problem

**LIST OF TABLES**

**Table Title page**

2.1: The output from a given graph 10

2.2. Summary of some data structure inventors 20

2.3: The difference in time complexity of various operations associated

with Binary heap, Binomial heap and Fibonacci heap. 34

**LIST OF FIGURES**

**Figure Title Page**

2.1: Procedure for the Dijkstra Algorithm 15

2.2: Pseudocodes for the Dijkstra Algorithm 16

2.3: Pseudocodes for the Binary Heap 23

2.4: Pseudocodes for the Binomial Heap Union (H1, H2) 27

2.5: Pseudocodes for the Fibonacci Heap CONSOLIDATE*(H)* 33

3.1 Proposed Framework for the Shortest Path Problem 48

**CHAPTER ONE**

**INTRODUCTION**

**1.1 Background to the Study**

The aspect of algorithms is an integral part of the theoretical computer science that has been in existence since the early days of the information age. It gave birth to many brilliant ideas used in solving fundamental computational problems. Therefore, algorithms are the solution to computational problems. They define methods that are used in solving problems that require the formulation of an algorithm for the solution.

An algorithm is a sequence of computational steps that transform the input into output or a sequence of steps or tools to solve computational problems. It is a well-defined computational procedure consisting of a set of instructions that takes some values or set of values as input, and produces some values or set of values, as output. In other words, an algorithm is a procedure that accepts data, manipulate them to follow the prescribed steps and to eventually fill the required unknown with the desired value(s) or a problem-solving formula that provides you with step-by-step instructions used to achieve a desired outcome (Kahneman, 2011). Algorithms have a vital and significant role in solving the computational problems. It can be refer to any well computational procedure that takes some value or set of value as input and produces some value or set of values as output (Panday 2015). Algorithm is a set of step by step procedures or a set of rules to follow for completing a specific task or solving a particular problem (Amy 2020). An algorithm is a procedure or formula for solving a problem based on conducting a sequence of specified actions (Rajan 2022).

Dijkstra’s algorithm (DA) is an efficient, exact method for finding the shortest paths between vertices in edge- and arc-weighted graphs. It is particularly useful in transportation problems when we want to determine the shortest (or fastest) route between two geographic locations on a road network (Rillet, 2006) It is also applicable in areas such as telecommunication, social network analysis, arbitrage, and currency exchange ( Lewis 2022).

Dijkstra`s algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, road networks (Edger, 1956). Dijkstra algorithm is used for finding the shortest path between two vertices in a graph. It is the single-source shortest path algorithm (i.e it can only be used to find the shortest path from a single source vertex to all other vertices. Dijkstra’s algorithm maintains a priority queue of vertices and repeatedly selects the vertex with the lowest distance from the source. The distance from each vertex updated as the algorithm progresses, and once the destination vertex is reached, the shortest path is found.

The algorithm exists in many variants. Dijkstra's original algorithm can find the shortest path between two given nodes, but a more common variant may fix a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.The most popular modifications include Dijkstra algorithm with Fibonacci heap, List, Indexed priority queue, Binary heap, D-way heap, Binomial heap andArray.

We will use Fibonacci heaps, Binary heap, and Binomial heaps to implement the priority queue needed in Dijkstra's algorithm.

Fibonacci heaps are implemented using a circular and doubly linked root list of heap-ordered trees in which each node contains a pointer to its parent and to one of its children. The children of each node are doubly linked in a circular linked list.

Binary heaps (BH) are types of priority queue (Min-heap: smaller priorities are earlier; Max-heap: are opposite) Based on heap-ordered trees (Min is at root) and an efficient representation of binary trees by arrays (No need for separate tree nodes and pointers (Eppstein 2023)

A Binomial heap is an extension of binary heap that provided faster union of merge operations provided by binary heap. A binomial heap is implemented by a set of binomial tree that satisfy binomial properties, Each binomial tree in a heap obeys the minimum-heap property; the key of a node is greater than or equal to the key of its parent. There can be at most one binomial tree for each other including zero order. (Geeks 2023)

Therefore, this research will present the performance and evaluate of Fibonacci heap, Binary heap, and Binomial heaps for shortest path problems. The goal is to determine the computer time, throughput, and computational time complexity for each algorithm to identify which variant of Dijkstra algorithm gives best performance.

**1.2 Statement of the Problem**

Several researchers have worked on means of determining shortest path problem of series of either directed or undirected graphs. Findings revealed that a research work was carried out on the single source shortest path problem which involved knowing the shortest path between a particular source vertex and target vertex. Also, research work was done on how to find the shortest path between two nodes in a network (Zhan 1998).

However, no work was done on determining the shortest path from a single source to all other reachable vertices coupled with the difficulty of their output to manage sharp edges. Hence, this work focuses on single source shortest path problem of labeled acyclic graph using three variants of Dijkstra algorithm to compare their performance interns of computer time, throughput and computational time complexity.

**1.3 Aim and Objectives**

The aim of this work is to carry out a performance evaluation of some selected variants of the Dijkstra Algorithm by using priority queue.

**The objectives are to:**

1. critically review three selected variants of Dijkstra algorithm
2. implement the three variants (Fibonacci heap, Binary heap, and Binomial heap) of Dijkstra algorithm.
3. compare the performance of the three-variant using Computer time, Throughput, and Computational time complexity.

**1.4 Significance of the Study**

Implementation of an algorithm will be more reliable and efficient in determining the performance evaluation of Dijkstra Algorithms. The significance of this research is to evaluate performance metric based on their influence. The study will help to reveal the most efficient variant of Dijkstra algorithms to be adopted when solving optimization problems,

**1.5 Scope of the Study**

This research will consider three (3) variant of Dijkstra’s algorithm which include Binary heap, Binomial heap, Fibonacci heap, and their performance metrics such as Computer time, Throughput and Computational time complexity within the operation shall be considered in evaluating the performance of the three algorithms.

**CHAPTER TWO**

**LITERATURE REVIEW**

**2.1 Shortest Path Algorithm**

The Shortest Path Problem (SPP) approach involves finding the shortest path between two vertices (or nodes) in a graph such as sum of weight of its constituent edges when minimized. 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 graph has its own properties which distinguishes the graph from one type to another because; the shortest path problem is solved base on the type of graph and its properties that are used. Different graph also require different methods of implementation. Therefore, to find ways or algorithms to solve the shortest path problem one should know the graph used. There are several variations of the shortest path algorithm (SPA), each designed to address different scenarios and constraints. Two of the most used variations are Bellman-Ford Algorithm and Dijkstra Algorithm (Mashitoh 2013).

**2.2 Bellman-Ford Algorithm**

The Bellman-Ford algorithm (BFA) can handle graphs with negative edge weights. It is particularly useful in scenarios where negative weights represent benefits or discounts. The algorithm iteratively relaxes the edges, gradually improving the estimated shortest path until convergence is achieved. Bellman-Ford can detect negative cycles, which are loops in the graph that reduce the path cost indefinitely, providing additional insights into the problem domain (Mashitoh 2013).

.

**2.3 A\* Algorithm**

The next common extension of Dijkstra is the A\* algorithm (AA), which is very similar to Greedy Best First Search, but uses both heuristic function and the actual distances between vertices.

The [A\* algorithm](https://en.wikipedia.org/wiki/A-star_algorithm) is a generalization of Dijkstra's algorithm that cuts down on the size of the sub graph that must be explored; if additional information is available that provides a lower bound on the "distance" to the target. A\* is one of the most popular and powerful short path algorithms and is often preferred to Dijkstra, as its use of heuristics can significantly increase the performance. The next well-known algorithm that relies on the Dijkstra algorithm is Johnson's algorithm. Johnson's algorithm uses Bellman-Ford algorithm to deal with negative weights and transform the input graph into one that can be processed by Dijkstra algorithm (Sneha 2017).

.

**2.4. Bi-Directional Dijkstra**

Bi-Directional Search (BDS), as the name implies, searches in two directions at the same time: one forward from the initial state and the other backward from the goal. The search stops when searches from both directions meet, and the optimal solution is proven. In many cases, it makes the search faster (Sneha 2017).

For example, in a search problem modeled by a tree with branching factor b and solution depth d. A Bi-directional search will expand 2b d/2 states instead of b d required by unidirectional search. Bi-directional Dijkstra algorithm is an example of this technique. 2 The heuristic search community traditionally uses Goal-directed search techniques (GST) to speed up Dijkstra algorithm.

Bi-directional Dijkstra Algorithm (BDA) computes lengths of shortest path from a start vertex to every other vertex in a weighted graph with non-negative weights. Bi-directional Dijkstra algorithm performs a bi-directional search with Dijkstra’s algorithm and can often reduce the practical running time for finding point-to-point shortest path as the name implies. This algorithm performs Dijkstra’s algorithm in both directions simultaneously, from the source towards the target and from the target towards the source. It stops when a node has been scanned from both directions. Searching from both the source and target in a homogenous graph can reduce the search space to half the size compared to only searching from the source .i. e. Unidirectional vs. Bidirectional search space. It is important not to assume that the shortest path always goes through the node where the forward and backward searches meet. It is obvious that the shortest path does not always go through the first node that has been scanned in both directions. The correct way of finding the shortest path using Bi-directional Dijkstra Algorithm is to consider which path is the shorter of the two options when the scan meet at a node (Ikeda 1994)

**2.5 Dijkstra's Algorithm**

Dijkstra’s algorithm (DA) is a popular choice when dealing with non-negative edge weights. It efficiently computes the shortest path from a single source node to all other nodes in the graph. The algorithm maintains a priority queue of nodes and iteratively selects the node with the smallest distance until it reaches the destination. Dijkstra Algorithm guarantees optimality, making it suitable for applications where finding the globally optimal solution is essential.

The Dijkstra algorithm belongs to a family of Best First Search Algorithms, (BFSA) and therefore used to find the shortest path between two points on a graph. Finding such a path in a graph may serve as an abstraction for solving real world problems such as finding the shortest route in road networks. The points or vertices in the graph may represent cities and the edges may represent roads connecting the cities. Each edge in Dijkstra algorithm is assigned a number, which represents distance of each route. Common services such as MapQuest can use this algorithm to find the shortest route between two points on a map. In a separate approach, network routing, the algorithm can be used to find the shortest path for data packets to take in a switching network. Another common use of Dijkstra algorithm and its variants can be found in the fields of Telephone and Flight networks. The algorithm that determines shortest paths from a source point or common vertex to all other points or vertices was introduced by the Dutch computer scientist, Edger W. Dijkstra. His work was discovered in 1956 and in1959.

According to Thomas *et al.* (2000), three problems involving shortest paths on arc-weighted graphs can be distinguished:

1. The single-source single-target shortest path problem (SSSPP) involves finding the shortest path between a particular source vertex s and target vertex t. In other words, we want to identify the s-t-path in G whose length (weight) is minimal among all s-t-paths.
2. The single-source shortest path problem (SSPP) involves determining the shortest path from a source s to all other reachable vertices in G. In this sense, we are seeking a “shortest path tree rooted at s.”
3. The all-pairs shortest path problem (ASPP): This involves finding the shortest path between every pair of vertices in G.

Dijkstra algorithm (DA) is used to solve the second problem in the above list. However, it can also be used for the three variants. To do this with the single-source single-target shortest path problem, there is a need to halt Dijkstra algorithm as soon as the target vertex becomes distinguished.

The Single Source Shortest Path Problem is a simple, common, but practically applicable problem in the realm of algorithms with real-world applications and consequences. It is the problem of finding the distance from one vertex in a graph to every other vertex in the graph. The Single Source Shortest Path Problem given in graph G = (V, A), To find a shortest path from a given source vertex s є V to each vertex v є V.

Let G = (V, A) be an arc – weighted, directed graph in which V is a set of n vertices, and A is a set of marcs (directed edges). In addition, let Ӷ(u) denote the set of vertices that are neighbours of a vertex u. that is, Ӷ(u) = {v, (u, v) є A}. Also define nonnegative weight (or length) w (u, v) for each arc (u, v) є A. the weight (or length of a path is by the sum of the weight of its arcs. (Thomas 2009)

To produce A shortest – path tree rooted at s, Dijkstra algorithm operates by maintaining a set D of so called “distinguished vertices.” Initially, only the source vertex s is considered distinguished. During execution, further vertices then are added to D. one at a time until all reachable vertices have been inserted. Two other data structures are also maintained. Fist a “label” L(u) is stored for each vertex u є V in the graph. During execution, L(u) stores the length of the shortest s – u path that uses distinguished vertices only. Consequently, on termination of the algorithm, L(u) gives the length of the shortest s – u path in the graph. If a vertex u has a label L (u) = ∞, then no s – u path is possible, then, a ‘predecessor’ P(u) is also stored for each vertex u є V. during execution, P (u) stores the vertex that occurs before u in the shortest s – u path (of length L(u) that uses distinguished vertices only. If a vertex u has no predecessor, then P(u) = NULL. On termination, these predecessor values can be used to construct the shortest paths from s to all reachable vertices.

**2.5.1. Simple statement of DijkstraAlgorithm**

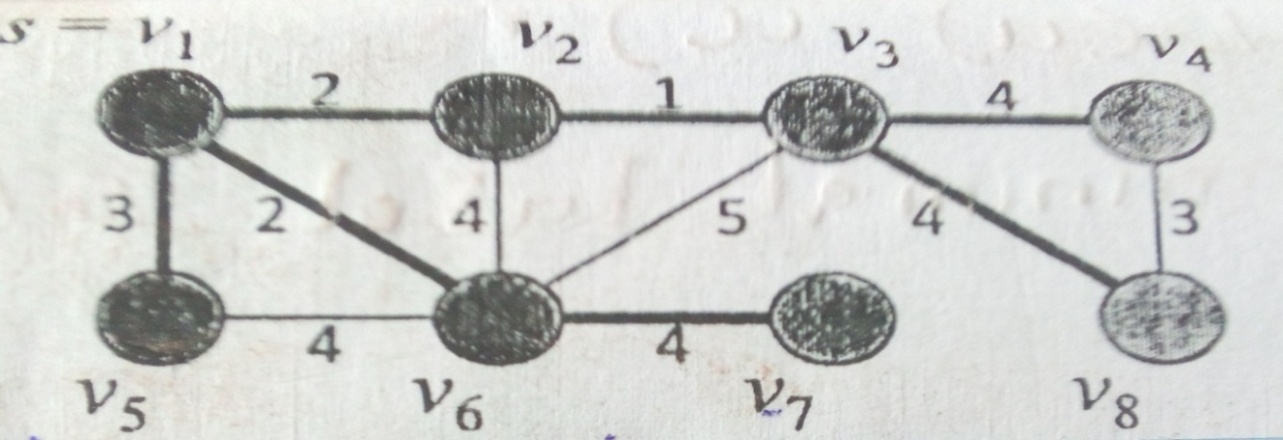
Input: an arc-weighted graph G = (V, A) and source vertex s є V

Output: A populated label array L and predecessor array P

1. Set L(u) = ∞ and P(u) = NULL for all u є V. also let D = Ǿ and set L(s) = 0
2. Choose a vertex u є V such that: (a) its value for L(u) is minimal: (b) L(u) is less than ∞; and (c) u is not in D. if no such vertex exists, then end; otherwise insert u into D and go to step 3.
3. For all neighbours v є Ӷ(u) that are not in D, if L(u) + w (u, v) < L(u) then set L(v) =

L(u) + w (u, v) and set P (v) = u, now return to step 2

For example, the output from the statement of Dijkstra algorithm using the indicated eight- vertex graph and source vertex s = v1 In figure 2.1 below..In this situation, the graph is an undirected graph. Note that, an arc (u, v) exists if and only if the arc (v, u) exists. In all cases, w (u, v) = w (v, u). The shortest path tree rooted at s (defined by P) is shown in bold lines in the graph. The output from the given example runs of this algorithm is shown in the following table

**Figure 2.1 the shortest path tree rooted at s shown in bold line in the graph**

**Table: 2.1: The output from a given graph**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 |
| L | 0 | 2 | 3 | 7 | 3 | 2 | 6 | 7 |
| P | NULL | V1 | V2 | V3 | V1 | V1 | V6 | V3 |

The Statement of Dijkstra algorithm describes three steps. In these steps, note that one vertex is inserted into D at each iteration. This gives 0(n) iterations of the algorithm in total. More so, within each iteration, there is a need to identify the vertex u ¢ D with the minimum label (an O(n) operation) and then examine (and possibly update) the labels of all vertices v є Ӷ(u).

Therefore, sequence of vertices that occurs in each shortest path starting at S is stored in P. the shortest S – U path (for all U є V) can be constructed the GET PATH Procedure

**Input:** The arc weighted graph G, predecessors P, source vertex *S,* and an arbitrary vertex *U*,

**Output:** A vertex sequence corresponding to the shortest *S-U* path in *G*

1. Let – (), and let v = u
2. if P (*u*) NULL then
3. while *vs*do
4. Append *v* to and set *v* = P (*v*)
5. Append *s* to and then reverse

The operation will start at u, and taking each preceding vertex until the source s is encountered, the S – U path is then reverse of this sequence. The shortest S – V8 – path from given example is written ᴫ = (S, V2, V3, V8). Because all arc weights are assumed to be nonnegative, therefore, paths returned by Dijkstra algorithm will be ‘simple.’ Meaning it cannot store the same vertex more than once (Lewis 2023).

In the real world, this problem is most often translated to finding the shortest distance between locations for land, water, and air travel, but it can also be used to simulate networking and layout problems, and state-based problems, like puzzles.

The most well-known algorithmic solution to this problem is Dijkstra's Algorithm, which uses a priority queue to keep track of the vertices that have been visited and their current distances from the source. It is in this priority queue where optimizations for this algorithm are typically made. One way to implement this priority queue is as a Fibonacci heap. The interactive visualizations here show how Fibonacci heap operations work, both as a standalone heap and when used in Dijkstra's Algorithm as a priority queue. Dijkstra’s Algorithm is used to find the shortest route from one vertex, called the source, to all others in a weighted graph, but it can also be adapted to focus the distance to one node, called a target.

**2.5. 2.Dijkstra Algorithm Procedures.**

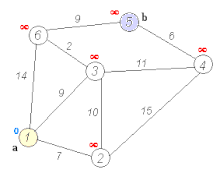
Let the distance of vertex Y be the distance from the source vertex to Y. Dijkstra algorithm will assign some initial distance values and will try to improve them step by step.

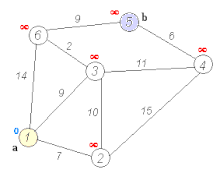
1. Mark all vertices unvisited. Create a set of all the unvisited vertices called the unvisited set.
2. Assign to every vertex a tentative distance value: set it to zero for our source vertex and to infinity for all other vertices. Set the source vertex as current.
3. For the current vertex, consider all its unvisited neighbors and calculate their tentative distances through the current vertex by adding the weight of the edge connecting the current vertex and neighbor to the current vertex's assigned value. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
4. When we are done considering all the unvisited neighbors of the current vertex, mark the current vertex as visited and remove it from the unvisited set. A visited vertex will never be checked again.
5. Once all reachable vertices are marked. You will then stop.
6. Otherwise, select the unvisited vertex that is marked with the smallest tentative distance, set it as the new "current vertex", and go back to step 3.

The "Randomize" button generates a semi-random graph with directed and weighted edges. The "Start" button steps through Dijkstra algorithm until all shortest paths are found. In the tool, the source vertex is initially marked in red. When a vertex is visited, it is marked in green.

**2.5.3. Algorithm Techniques**

Dijkstra algorithm: to find the shortest path between a.and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Usually used with priority queue or heap for optimization (Freedman 1987).



The algorithm exists in many variants. Dijkstra original algorithm can find the shortest path between two given nodes, but a more common variant may fix a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. For a given source node in the graph, the algorithm finds the shortest path between that node and every other.  It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and costs of edge paths represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), then Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a subroutine in other algorithms such as Johnson's.

The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalized to use any labels that are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the generic Dijkstra Shortest-Path Algorithm.

**2.5.4. Techniques of operation of Dijkstra Algorithm**

There are several ways of implement Dijkstra’s algorithm, but the most common ones are.

1. Using priority queue to keep tracts of all vertices.
2. Using an array to keep tract of Distances.
3. Using a set to keep tract of the visited vertices.

Dijkstra Shortest Path Algorithm (DSPA) using priority queue (Heap). In each graph and a source vertex in the graph, finding the shortest paths from the source to all vertices in the given graph. The algorithm will base on the following steps.

**Table 2.1.: Procedure for the Dijkstra Algorithm**

1. Initialize distances of all vertices as infinite.
2. Create an empty.

Priority-queue **Pq.** Every item of Pq is a pair (weight, vertex). Weight (or distance) is used as first item of pair, as first item is by default used to compare two pairs.

1. Insert source vertex into Pq and make its distance as O.
2. While either Pq does not become empty.
3. Extract minimum distance vertex be U.
4. Loop through all adjacent of U and do the following for every vertex V.
5. If there is a shorter path to V through U.
6. update distance of V. i.e, do dist.[v] =

Dist.[u] + weight (u, v).

1. Insert v into Pq. (Even if V is already there).
2. Print distance array dist. []; to print all shortest paths.

**Table 2.2: Pseudocodes for the Dijkstra Algorithm (Rhyd 2023)**

***Input:*** *An arc weighted graph G = (V, A), and source vertex s є V*

***Output****: A populated label array L and predecessor array P*

1. *For all u є V, set L(u) = ∞, set D(u) = false, and set P(u) = NULL*
2. *Set L(s) = 0 and insert the ordered pair (Ls), s) into Q*
3. *While Q is not empty do*
4. *Let (L(u), u) be the element is Q with the minimum value for L(u)*
5. *Remove the element (L(u), u) from Q.*
6. *Set D(u) = true*
7. ***Foreach****vєӶ(u) such that D(v) = false do*
8. *If L(u) + w (u, v) < L(v) then*
9. *If L(v) < ∞ then*
10. *Decrease the key of (L(u), v) to L(u) + w (u,v). That is replace the element (L(v), u) in Q with the element (L(u) + w (u, v), v).*
11. *Else*
12. *Insect the element (L(u) + w (u, v), v) into Q*
13. *Set L(v) = L(u) + w (u, v) and set P(v) = u*

**2.6. Variants of Dijkstra Algorithm**

Dijkstra may be viewed as one of the most fundamental algorithms for solving the shortest path problem. There are numerous variants and extensions of Dijkstra algorithm. Variants of Dijkstra algorithm is known as Heap. Heap is exactly a data structure that allows us to store the set of edges and efficiently retrieve the one with minimum cost. A heap is a specialized data structure (usually tree-based) that satisfies the heap property: If B is a child node of A, then key(B) ≥ key(A). The heap is one the most efficient implementation of an abstract data type called a priority queue (Radek 2013).

The operations commonly performed with a heap are.

1. Insert (x); add a new key x to heap.
2. Access Min; finds and returns the minimum item of the heap.
3. Delete Min; removes the minimum node of the heap (usually, the minimum node is the root of a heap).
4. Decrease Key (x, d); decrease x key within the heap by d.
5. Merge (H1, H2); joins two heaps H1, and H2 to form a valid new heap containing all the elements of both.
6. Delete (x): removes a key x of a heap.

The items and their keys will be stored as nodes in a collection of heap-ordered trees. A heap-ordered tree is a rooted tree where the key of any node is no less than the key to its parent. Moreover, Dijkstra can be also often modified to solve various problem-specific tasks,

Johnson's implementation uses Fibonacci-heap min priority queue instead of a Binary Heap, which yields not only exposure to a larger set of problems, but also a better performance.

When other heaps were used to replace Fibonacci heap, different time complexities were obtained. In 1990, when radix heap was introduced to use with Dijkstra algorithm, time complexity obtained was 0(m+nlogc) where C was maximum edge cost. When some modifications were made to the radix heap with two level form, the complexity of executing Dijkstra algorithm became 0 (M+). When radix heap was combined with Fibonacci to solve Dijkstra algorithm the performance obtained was 0 (M+ ). The complexities obtained showthat using different data structuresresulted in different runtime.

As mentioned earlier the most common implementation of Dijkstra algorithm uses binary heap as its underlying data structure. Following the binary heap is a Leftist heap, developed by Clark Allah Crane in 1972. Binomial heap (BH) was developed by Jean Vullemin in year 1978 to supports all the heap operations in 0(Logn) worst-case times per operation leading to amortized cost analysis is Fibonacci heap which was introduced in 1987 by Freedman and Tarjan .In this heap, insert and decrease-key operation done in 0(1) amortized time and delete-min in 0(logn) amortized time. However, this heap has its limitations. As practical matter this heap is not efficient; it is also hard to implement as the structure if this heap is complicated (Gerth2012).

The skew heap 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 0(logn) time (Daniel 1986)

Driscoll introduced Relaxed heap in 1988. This is the first heap that allow the heap order to be violated, that means 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 2-3 heap was introduced in 1999 by Takaoba. 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 (Tadao 20003). A year later, Trinomial heap was introduced that supports the decrease-key operation in 0 (1) worse case time. This heap employs the ideas of a bad child or Violation node introduced in the relaxed heap 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. With 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, but Freedman provided analysis for the decrease-key operation as n(loglogn). (Michael 1999). This analysis however was reviewed by Pettier in 2005 and He proved that the decrease-key operation was done in 0(2). A small modification was made to the pairing heap, with the modification, Elmasry 2009 gave 0(loglogn) for the decrease-key operation.

Other heaps that have the same complexity as Fibonacci heap are Thin and Thick heap and Quake heap. Heaps recently developed include the Rank-pairing heap, Violation heap and Strick heap. With any other data structures such as Fibonacci, 2-3 heap or Trinomial heap, the running time depends on how the operations of delete-min and decrease-key are performed e.g if the delete-min. takes 0 (logn) time and decrease-key is in 0(1) time, then the total running time is 0(m+nlogn) time. Note that, expected number of decrease-key operation when solving the shortest path problem is 0(nlog) (Noshita 1985).

Therefore, this research will focus on three of the most famous variants which are Binary heap, Binomial heap and Fibonacci heap of Dijkstra. Algorithm

**Table:2.3. Summary of some data structure inventors (Mashitoh 2013).**

|  |  |  |
| --- | --- | --- |
| Year | Author | Heaps |
| 1964 | William | Binary heap |
| 1972 | Crane | Liftist heap |
| 1978 | Vuillemin | Binomial |
| 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, Sedyewick, Sleator and Tarjan | Pairing heap |
| 2000 | Takaoka | Trinomial heap |
| 2008 | Kaplan and Tarjan | Thin and Thick heap |
| 2009 | Chan | Quake heap |
| 2010 | Elmasry | Violation heap |
| 2012 | Brodal, Lagogiannis and Tarjan | Strict Fibonacci Heap |

**2.6.1 Binary Heap**

A binary heap is a complete binary tree which is used to stored data efficiently either to get the Max or Min element based on its structure.

A binary heap is either Min heap or Max heap. For example, in Min. Binary heap, the key at the root must be minimum among all keys present in Binary heap. The same property must be recursively true for all nodes in binary tree. Max. Binary heap is like Min. heap. (Prashant Srivastava 2020).

**2.6.2 Operation on heaps**

Below are some standard operations on min. heap.

1. Get Min (); it returns the root element of Min heap. The time complexity of this operation is 0(1). In case of a Max heap, it would be getting Max ().
2. Extract Min (); Removes the minimum elements from Min heap. The time complexity of this operation is 0(logN) as this operation need to maintain the heap property [by calling heapify ( ) } after removing the root.
3. Decrease key ( ): decreases the value of the key. The time complexity of this operation id 0(logN). If the decreased key value of a node is greater than the parent of the node, then we do not need to do anything, otherwise, we need to traverse up to fix the violated heap property.
4. Insert ( ): inserting a new key takes 0(logN) time. We add a new key at the end of the tree. If the new key is greater than its parent, then we do not need to do anything, otherwise, we need to traverse up to fix the violated heap property.
5. Delete ( ): deleting a key also takes 0(logn) time. We replace the key to be deleted with the minimum infinite by calling decrease key ( ). After decreasing key ( ). The minus infinite value must reach root, so we call extract Min ( ) to remove the key.
6. Build Heap: 0(n.) = 0(n)
7. Merge: 0(n) by building a new heap. (Jiri 2013)

**Table 2.4: Pseudocodes for the Binary Heap (HEAPIFY) (Jiri el tol 2013).**

1. *If Left(i) ≤ heapsize and A[left(i)] < A[i].*
2. *Smallest<− left(i)*
3. *Else*
4. *Smallest <− i*
5. *If Right (i) ≤ heapsize and A [Right (i)] < A [smallest]*
6. *smallest<− Right(i)*
7. *If smallest ≠ i*
8. *Swap A[i} and A[smallest]*
9. *Heapify(smallest).*

**2.6.3 Applications of heap**

i. Heap sort uses Binary heap to sort an array in 0 (nlogn) times. Priority queue (PQ) can be efficiently implemented using Binary heap because it supports insert (), delete (), and extract Max ( ), decrease key ( ) operations in 0(logn) time.

ii. Binomial heap and Fibonacci heap are variations of Binary heap. These variations perform union also efficiently.

1. Graph algorithms: the priority queue is especially used in graph algorithms like Dijkstra’s shortest path and prim’s minimum spanning tree.

Many problems can be efficiently solved using heaps. (GeekforGeek 2020).

**2.7 Binomial heap**

Binomial heap (BH) is an extension of binary heap that provides faster union of merge operations provided by binary heap. Binomial heap is a collection of binomial trees**.** A binomial tree of order has one node. A binomial tree of order K can be constructed by taking two binomial trees of order K-1 and making one the leftmost child of the other.

A binomial tree of order K has the following properties.

1. It has exactly 2knodes
2. It has depth as K.
3. There are exactly Kaic, nodes at depth I for i= 0, i………k.
4. The root has degree k and children of the root are themselves Binomial trees with order K-1, K-2 -----0 from left to right.

**2.7.1 Operation of Binomial heap**

The main operation of binomial is a union (), the union operation is to combine two binomial heaps into one. Let us discuss other operations first.

Other operations

1. Insert (H, K); insert a key K to Binomial heap “H.” This operation first creates a Binomial heap with a single key ‘K,’ then call union on H and the new Binomial heap.
2. Getting (H); A simple way to get in () is to traverse the list of the roots of binomial Tree and return the minimum key the implementation requires 0(logn) time. It can be –optimized to (1) by maintaining a pointer to the minimum key root.
3. Extracting (H); this operation also uses a union () we will first call get Min ( ) to find the minimum key Binomial tree, then remove the node and create a new Binomial heap by connecting all sub trees of the removed minimum node. Finally, we will call union ( ) on H and the newly created Binomial Heap. This operation requires 0(logn) time.
4. Delete (H). Like binary heap, the delete operation first reduce the key to minus infinite, then call extracting ().
5. Decrease key (H), decrease key () is also like Binary Heap. We will compare the decreased key with its parent and if the parent’s key is more, we swap key and recur for the parent has a smaller key or we hit the root node. The time complexity of the decrease key () is 0( logn).

**2.7.2 Union operation in Binomial Heap**

Given the Binomial heaps. H1 and H2, Union (H1, H2) create a single Binomial Heap.

1. The first step is to simplify merge the two Heaps in non-decreasing order of degrees.
2. After the simple merge, we need to measure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Tree of some order. We traverse the list of merged roots, we keep track of three pointer, Prev. X, and next X. when traverse the list of roots we will consider the four cases.

Case1: Order of X and next X are not the same, we simplify move ahead. In the following three cases: order of X and next-X are the same.

Case 2: if the order of next-next X is also the same move ahead.

Case 3: if the key to X is smaller than or equal to the key of next-X then, make next-x a child of X by linking it with X

Case 4: if the key to X is greater, then make X the child of next.

**Table 2.5: Pseudocodes for the Binomial *Heap Union (H1, H2) (Abhinau 2013)***

1. *H <− MAKE BINOMIAL- HEAP ()*
2. *Head(H) <− BINOMIAL – HEAP MAERGE (H1, H2)*
3. *Free the objects H1 and H2 but not the lists they point to*
4. *If head [H] = NIL*
5. *then return H*
6. *Prev x <− NIL*
7. *X <− head [H]*
8. *Next x <− sibling [x]*
9. *While next –x ≠ NIL*
10. *.do if (degree [x] ≠ degree[ next –x]) or*

*(sibling [next-x] ≠ NIL*

*And degree (sibling [next – x]) =degree [x])*

1. *Then prev –x <− x*
2. *x<− next – x*
3. *Else if key [x] ≤ key [ next – x]*
4. *.then sibling[x]<− sibling (next . x)*
5. *BINOMIAL\_LINK (next .x, x)*
6. *Else if prev –x = NIL*
7. *.then head[H} <−next.x*
8. *Else sibling [prev – x] <− next – x*
9. *BINOMIAL – LINK ( x, next – x)*
10. *X <− next x*
11. *.next – x <−sibling [x]*
12. *Return H.*

**2.8 Fibonacci Heap**

To speed up Dijkstra Algorithm for the single-source shortest path problem with nonnegative length edges, Fredman, and Tarjan in 1987 developed the Fibonacci heap.

The Fibonacci heap (FH) has similar properties to the binomial heap, such that every binomial heap is a Fibonacci heap, but not the other way around. The Fibonacci heap can contain multiple trees consisting of single nodes, while the binomial heap always consists of trees with 2n nodes, where each tree has a different n value. This is a result of the Fibonacci heap not merging singular nodes into trees when inserting elements, to keep the insertion time constant, unlike the binomial heap which eagerly merges every insertion into the tree. The subsections below explain the three most common heap operations in the Fibonacci heap.

**2.8.1 Insertion**

When inserting elements into an empty Fibonacci heap, it starts out as a doubly linked list of root-nodes with a pointer saved to the minimum element. The heap will remain a doubly linked list of unordered nodes until the minimal node is extracted from the heap.

**2.8.2 Extract-min**

The extraction of the minimum node is a simple procedure, where the children of the minimum node are added to the root-list of the heap and their parent 2 pointer is removed. The minimum node is linked out from the double linked root list, and the pointer to said node is returned.

**2.8.3 Consolidation**

Although consolidation is not a common heap operation. It is a crucial help function for the Fibonacci heap by extracting the minimum element results in a consolidation of the root-nodes in the heap into trees. Each node in the tree can have a maximum of log N children, where N is the total amount of nodes in the tree (Cormen 2009). Each parent will have a single pointer to one of its children, while all children will point to its parent. The child nodes will also keep pointers to two other children, the left and right one, resulting in a doubly linked list between the children. Since the nodes are not stored in any order in the root list, the time to find the new minimum scales with the number of trees in the root list. To reduce this time in future operations, the consolidation will merge trees with the same degree, where the degree represents the number of children of the root-node in each tree. The tree with the smallest root-node will adopt the root-node of the other tree, merging the two trees into a tree with previous degree plus one. Then, it is possible for the root-list to contain multiple trees even after consolidation if none of the trees in the root-list are of the same degree. (Elwira 2020).

Fibonacci heap (FH) are sometimes referred to as “Easy” data structures because their structure is less strictly managed as they perform operations than in many other data structures. But this “Easiness” allows them to have extremely fast amortized time complexities for common heap operations.

|  |  |
| --- | --- |
| Operation | Time Complexity |
| insert () | O (1) |
| ReportMin() | O (1) |
| DeleteMin() | O (log n) amortized |
| Delete () | O (log n) amortized |
| Decrease Key () | O (1) amortized |
| Merge () | O (1) |

Fibonacci heap supports the same operations but have the advantages that operations that do not involve deleting an element (Access Min, Merge, and Decrease Key) run in O (1) amortized time. Operations Delete and Delete Min have 0(log(*n*)) amortized time complexity(Jiri 2013).

Fibonacci heap is implemented using a circular and doubly linked root list of heap-ordered trees in which each node contains a pointer to its parent and to one of its children. The children of each node are doubly linked in a circular linked list. There is always a pointer to the node with the minimum key in the heap. It is this high connectivity between nodes that allows 0(1) time insertion, minimum-reporting, key-decreasing and merging.

First, hit the "Initialize Heap" button to set up the Fibonacci heap. From there, enter numerical values in the input box and then hit the "Insert" button to insert them into the heap. The "Delete Minimum" button removes the minimum value from the heap and the "Delete Key" button deletes the value in the input box from the heap if it exists.

Dijkstra original Shortest Path Algorithm does not use a priority queue and runs in 0(V2) time. When using a Fibonacci heap as a priority queue, it runs in 0(E + V log V) time, which is asymptomtically the fastest known time complexity for this problem. However, due to their programming complexity, and for some practical purposes, Fibonacci heap is not always a necessity or make a significant improvement. Fibonacci heap are best reserved for applications in which the number of deletion is small compared to the use of other operations. In further iterations of this demo, the efficacy of different priority queue implementations could be tested with timing tests for graphs of different sizes and properties. For now, animation overhead erase any improvements made by the Fibonacci heap, and this demo solely shows a visualization of the two structures in action. (Kennedy Bailey)

First, hit the "Randomize" button to generate a semi-random graph and read its data into the Fibonacci heap. From there, hit the step button to step through the Dijkstra Algorithm to find the shortest path from a source node (marked in red) to one specific target node (marked in blue) and watch the changes in the Fibonacci heap reflected in its visualization.

**2.8.4. Some Interesting facts about Fibonacci heap.**

1. The reduced time complexity of decrease –key has importance in Dijkstra and Prim algorithms. With binary Heap, the time complexity of these algorithms is 0(Vlogv+Elogv). If Fibonacci heap is used, then the time complexity is improved to 0(VlogV+E).
2. Although Fibonacci heap looks promising time complexity-wise, it has been found to be slow in practice as hidden constants are high.
3. Fibonacci heap is called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci heap has a degree at most 0(logn) and the size of a sub-tree rooted in a node of degree K is at least Fk+2 where Fk is the Kth Fibonacci n number.

**2.8.5 Advantages of Fibonacci heap**

1. Fast amortized running time: the running time of operations such as insert, extract-min and merge in a Fibonacci heap is 0(1) for making it one of the most efficient data structures for these operations.
2. Efficient memory usage: Fibonacci heap have a relatively small constant factor compared to other data structures, making them a more memory efficient choice in some applications.
3. Lazy consolidation: the use of lazy consolidation allows for the merging of trees to be performed more efficiently of the batches, rather than one at a time improving the efficiency of the mergedoperations. (GFGWRITEBOT 2023)
   * 1. **Disadvantages of Fibonacci heap**
4. Increased complexity: the structure and operations of a Fibonacci heaps are more complex than those of binary or binomial heaps making it a less imitative data structure for some users.
5. Less well-known; compared to other data structures, Fibonacci heap are less popular and widely used, making it more difficult to find resources and support for implementation and optimization. (Pradesh 2013)

**Table 2.6: Pseudocodes for the Fibonacci Heap*CONSOLIDATE(H) (Elwira 2020)***

1. *Let A[0----D(H.n)] be a new array*
2. *For I = 0 to D(H.n)*
3. *A(i)= Nil*
4. *For each node w in the root list of H*
5. *X = w*
6. *D = x degree*
7. *While A[d] ≠ NIL*
8. *Y = A [d] // another node with the same degree as x*
9. *If x. key > y. key*
10. *Exchange x with y*
11. *FIB.HEAP-LINK (H, y, x )*
12. *A(d)* ***= NIL***
13. ***d****= d+i*
14. *A[d] = x*
15. *H.min = NIL*
16. *For I = 0 to D (H. n)*
17. *If A[i] ≠ NIL*
18. *If H. min = = NIL*
19. *Create a root list for H containing just A[i]*
20. *H. min = A[i]*
21. *Else insert A[i] into H’s root list.*
22. *If A[i]. key,H.min.key*
23. *H.min = A[i]*

**Table 2.7: The difference in time complexity of various operations associated with binary heap, Binomial heap and Fibonacci heaps are mentioned in the following table. (By Prashant 2020)**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Binary heap | Binomial heap | Fibonacci heap |
| Insert | 0(logn) | 0(logn) | 0(1) |
| Find-min | 0(1) | 0(logn) | 0(1) |
| Delete | 0(logn) | 0(logn) | 0(logn) |
| Decrease-key | 0(logn) | 0(logn) | 0(1) |
| Union | 0(n) | 0(logn) | 0(1) |

In terms of time complexity, Fibonacci heap beats both binary and binomial heaps. Below are amortized time complexities of Fibonacci heap:

* 1. Find min 0(1) (Same as both binary and binomial heap
  2. Delete Min 0(Log n) [0(Log n) in both binary and binomial heaps.
  3. Insert 0(1) (0Log n) in binary and 0(1) in binomial heap.
  4. Decrease key: 0(1) (0log n) in both binary and binomial heaps.
  5. Merge: 0(1) [0(mlogn) or 0 (m + n) in binary and0(log n) in binomial]

**2.8.7 Amortized analysis**

An amortized analysis is the time required to perform a sequence of data-structure operations is average over all the operations performed. Amortized analysis can be used to show the average cost of an operation when small. If an average over a sequence of operations is performed, though a single operation within the sequence might be expensive. An amortized analysis guarantees the average performance of each operation in the worst case. (Radek 2013)

**2.9 Performance Metrics Analysis**

Performance metrics are data used to track process within a business. These can be achieved using activities of the employee behaviour and productivity as key metrics. Employers use these metrics to evaluate performance. For Dijkstra algorithm using matrix representation, the time complexity is 0(V^2) but, for list representation the time complexity is 0((V+E) Log(V)).

To implement Dijkstra algorithm, we need to initialize three values;

1. dist.- An array of the minimum distance from the source node S to each node in the graph. At the beginning dist. (S)=0 and for all other node V, dist. (V) = ∞. This dist. array will be re-calculated and finalized when the shortest distance to every node is found.
2. Q- A priority queue of all nodes in the graph. At the end of the process, Q will be empty.
3. S- A set to indicate which nodes have been visited by the algorithm. At the end of the process S will contain all the nodes of the graph

**2.9.1** **Computer time**

Computer time is the amount of time requires by task to completes its execution. The execution time or CPU time of a provided task is defined as the time spent by the system in executing that job, including the moment invested in executing run-time or system service on its behalf. Measuring the total time elapsed to execute the code block in seconds, milliseconds, minutes and hours and get the execution time of functions and loops or measures the code performance, we need to calculate the time taken by the script/ program to execute. Measuring the execution of a program or parts of it depends on your operating system and version.

**2.9.1.1 Time functions**

To measure the time required to run a function one can use the “time it function”. The “time it functions” calls the specified function multiple times and returns the median of the measurement. It takes a handle to the function to be measured and returned. Suppose that you have define a function, computefunction, that takes two inputs x and y, that are defined in your workspace, you can compute the time to execute the function using “time it”.

F =@() mycomputefunction ( x, y )

Time it (f).

**2.9.1.2 Time portions of code**

To estimate how long a portion of your program takes to run or to complete the spaced of different implementations of portions of the program, use the stopwatch timer functions tic and toc. Invoking tic starts the timer and the next toc reads the elapsed time

Tic

% the program section to t.

Toc

Sometimes programs run too fast for tic and toc to provide useful data. If your code is faster than 1/10 second, consider measuring it running in a loop and then average to find the time for a single run.

**2.9.1.3 Performance of timing functions**

The time it functions can be used for a rigorous measurement of function execution time and the stopwatch timer functions tic and toc, enable you to time how long your code takes to run. It also used to estimate time for smaller portion of code that are not complete functions.

The performance of your code such as function call information and execution time to individual lines of code can be done by the use MATLAB.

**2.9.1.4 Differences between CPU time function, tic/toc and time it**

It is recommended that you use time it or tic and toc to measure the performance of your code, this function return wall-clock time. Unlike tic and toc, the time it functions calls your code multiple times and therefore, considers first-time cost.

The CPU time function measures the total CPU time and sums across all threads. This measurement is different from wall-clock time that time it or tic/toc return, and could be misleading for example; the CPU time for the pause function is typically small, but the wall-clock time accounts for the actual time that MATLAB execution is paused. Therefore, the wall-clock time might be longer, if your function uses four processing cores equally, the CPU time could be approximately four times higher than the wall clock time.

**2.9.2 Throughput**

Throughput or flow rate is a calculation that is commonly made in operations management that allows managers to see what output is in each amount of time. This output can be in either a product or service environment. The formula is based on little law which is used to calculate the average number of something over a given amount of time. When the variables are arranged properly you get the formula for throughput.

TH = I/T

1. TH: - is the throughput that we are determining or the average output of something over a given amount of time. The time is most illustrated per minute, hour, or day.
2. I: - is the inventory that used over a period. This can be physical inventory or service inventory, where the inventory is the customer.
3. T: - is the total time required to complete the inventory or task. (Brain 2023).

For example: a soap manufacturer wants to knowquantity of soap it produces per hour. It knows it produces ten thousandbars of inventory per day and that its machines run for first and second shift which are 16 hours T per day. It can calculate throughput as.

TH = 10000/16

TH = 625 per hour

Quantity of soap producing per minutes 1h = 60 minutes

625/60 = 10.4bar produced each minute of operation.

**2.9.2 Computational Time complexity**

There are multiple ways we can implement this algorithm, each way utilizes different data structures to store the graph as well as the priority queue. The differences between these implementation lead to different time complexities. We will discuss the time complexities of two main cases for Dijkstra implementations.

Case one

1. The given graph G = (V, E) represents an adjacency matrix. Here w (u, v) stores the weight of edge (u, v).
2. The priority queue Q represents an unordered list. Let /E/ and /V/ be the number of the edges and vertices in the graph, respectively, then the time complexitycalculated.
3. Adding all /v/ vertices to Q takes 0 (/v/)time.
4. Removing the node with minimal *dist*. takes 0(/v/) time and we only need 0(1) to recalculate dist.(u) and update Q. in case we use an adjacency matrix we will need to loop for /v/ vertices to update the distance array.
5. The time taken by each of the iteration of the loop is 0(/v/), as one vertex deleted from Q per loop.
6. Thus, total time complexity becomes 0(/v/) + 0(/v/) x 0(/v/) = 0(/v/2).

Case 2:

1. The given graph G = (V, E) is represented as an adjacency list.
2. The priority queue Q is represented as a binary heap or a Fibonacci heap.

The time complexity using a binary heap;

In this case time complexity is.

1. It takes 0 (/v/) time to construct the initial priority queue of /v/ vertices.
2. With adjacency list representation, all vertices of the graph can be traversedusing BFS. Therefore, iterating over all vertices neighbours and updating their *dist.*values over the course of a run of the algorithm takes 0(/E/) time.
3. The time taken for each of the iteration of the loop is 0(/v/) as one vertex is removed from Q per loop.
4. The binary heap data structure allows us to extract min (remove the node with minimal dist and update an element (recalculate dist. [u]) in 0(log/v/) time.
5. Therefore, the time complexity becomes; 0(/E/) + 0 (/E/ x log/v/) + O (/V/ x log/v/) which is 0 ((/E/ +/V/) x log/v/) = 0 (/E/ x log /v/). Since /E/ ≥ /V/ - 1 as G is a connected graph (Baeldung 2022)

Time complexity using a Fibonacci heap;

The Fibonacci heap allows us to insert a new element in 0 (1) and extract the node with minimal dist in 0 (log/v). Therefore, the time complexity will be.

1. The time taken for each iteration of the loop and extract min is 0(/v/) as one vertex is removed from Q per loop.
2. Iterating over all vertices neighbours and updating their dist. values for run of the algorithm takes O (log /v/) time, the total of all dist. calculation and priority value updates takes O (/E/ x log/v/) time.
3. The overall time complexity becomes 0(/v/ + /E/ x log /v/) (Baeldung 2022).

**2.10 Applications of the Shortest Path Algorithm**

**Network Routing**

One of the most prevalent applications of the shortest path algorithm is in network routing. In this context, the graph represents a network infrastructure, and the edges correspond to connections between nodes or routers. By finding the shortest path, network routers can efficiently determine how to forward data packets from the source to the destination, optimizing the overall network performance.

**Transportation Logistics**

The transportation industry heavily relies on the shortest path algorithm to optimize routes for vehicles, such as delivery trucks or ride-sharing services. By considering factors like road conditions, traffic congestion, and distances between locations, the algorithm can calculate the most time-efficient or cost-effective routes. This enablescompanies to minimize fuel consumption, reduce delivery times, and overall streamlining their operations.

**Social Network Analysis**

Shortest path algorithms are also invaluable in social network analysis, where the graph represents relationships between individuals or entities. By analyzing the shortest paths, data scientists can uncover influential individuals, identify communities, or understand the flow of information within a network. This information can be leverage for targeted advertising, recommendation systems, or understanding the spread of information or diseases in a population.

**Recommendation Systems**

Recommendation systems, such as those used by online platforms like Netflix or Amazon, rely on the shortest path algorithm to suggest relevant items to users. By analyzing user preferences and item similarities, the algorithm identifies the shortest path between the user and the items they might find interesting. This approach helps improve user engagement, increase sales, and enhance the overall user experience.

**2.11 Importance of Dijkstra Algorithm**

1. To determine the quickest route.
2. It’s useful in Google maps.
3. It can be employed to identify the shortest path.
4. It is extremely popular in the Geographical maps.
5. It used to locate the points on the map that correspond to the graph’s vertices.
6. To identify the open shortest path first, Dijkstra algorithm required.
7. It is needed in I.P routing.
8. The telephone network makes use of it.
9. It is also used to improve the efficiency of a computer program.

**2.12 Review of Related Works**

[Freedman and Tarjan (1984](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#CITEREFFredmanTarjan1984)) proposed [Fibonacci heap](https://en.wikipedia.org/wiki/Fibonacci_heap) using min-priority queue to optimize the running time complexity to θ (|E|+|V|\log |V|)}= θ (|E|+|V|\log |V|)}. Θ(|�|+|�|log⁡|�|)This is [asymptotically](https://en.wikipedia.org/wiki/Asymptotic_computational_complexity) the fastest known single-source [shortest-path algorithm](https://en.wikipedia.org/wiki/Shortest_path_problem) for arbitrary [directed graphs](https://en.wikipedia.org/wiki/Directed_graph) with unbounded non-negative weights. However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can indeed be improved further as detailed in [specialized variants](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Specialized_variants). Additionally, if preprocessing is allowed contraction algorithms such as [hierarchies](https://en.wikipedia.org/wiki/Contraction_hierarchy) can be up to seven orders of magnitude faster.

Ravindra (1993); proposed weather Single source shortest path can be solved in 0(m+nloglogc) has been solved by Mikkel (2003). He shows that with integer priority queue that performs decrease-key operation in constant time, the Single source shortest path problem can be solved efficiently, that is in 0(m+nloglogc) time.

. Freedman (1999): presented three algorithmically related data structures for implementing moldable priority queues (IMPQ). Binomial heap, Fibonacci heap and Pairing heap, what three algorithms have in common is that (1) they are comprised of heaps ordered by tree (2) the comparisons performed to execute extract min. operations exclusively involve keys stored in the roots of three and (3) common side effect of a comparison between two root keys is the linking of the respective roots. One tree becomes a new sub-tree joined.

Santoso (2010); analyzed the performance of Dijkstra, using A\* and Ant algorithm for finding optimal path, A\* and Ant algorithm when finding optimal path of certain route. The result shows that Ant algorithm is not suitable for the path finding algorithm because it is less stable and requires a long time to do a search while Dijkstra and A\* algorithm provide optimal results in a quick time.

Abhinau (2013); Library1 Providing data structures (LPDS) Viz-a-Viz Binomial heap and Fibonacci heap described with primitive operations, performance and usage, comparison with an existing Library for binary heap in Racket added to observe any run time improvement in practice for operations which have similar run-time bounds in its theory.

Buyue (2016), presented the Performance Evaluation (PE) of A\* Algorithms, the used of A\* algorithms, which over the years has had a lot of variations, the thesis find how Dijkstra algorithm, IDA\*, Theta\* and HPA\* compare against A\* based on the variable’s computation time, number of opened nodes, path length as well as number of path nodes. A\* and way they specifically improve A\* an experiment was set up where the algorithms were implemented and evaluated over several maps with varying attributes of the result of the tested algorithms for computation time, number of opened nodes, path length and number of path nodes over several different maps as well as the average performance overall maps.

Martell and Sandberg (2016); evaluated the performance of A\* algorithm; the thesis find how Dijkstra algorithm IDA\* Theta\* and HPA\* compare based against A\* based on the variable’s computation time, number of opened nodes, path length as well as number of path nodes. The result shows that A\* perform well overall, with Dijkstra algorithm trailing shortly behind in computation time and expanded nodes, Theta\* find the best path with overall good computation time married by few spikes on large open maps. HPA\* perform poorly overall when fairly compared but has by far the computation time and node expansion when partially pre-computed. IDA\* find the same paths as A\* and Dijkstra algorithm but has a notably worse computation time than the other algorithm and should be avoided on octal grid maps.

Enech and Arinze (2017); Carried out Comparative Analysis and Implementation of Dijkstra shortest path algorithm for emergency response and logistic planning Dijkstra algorithm implemented with double bucket dynamic data structure is selected for implementing the proposed route planning system as past research efforts has proven that it is the fastest with run-time improvements from 0(m+n/logc) to 0(m) respectively.

Sawlani (2017): Explaining the performance of Bi-directional Dijkstra and A\* on Road network. The thesis investigates the performance of A\* and Bi-directional Dijkstra in road networks to see how they compare and to see if there is a principled explanation for the different approaches. analysis reveals why Bi-directional Dijkstra can performed well in this domain but also show simple mistake that can be made when building test problems that hurts the performance A\*.

Adeolu (2018): presented the link state algorithms make use of the Dijkstra shortest path algorithm to perform the function of its any-to-any path finding. The algorithm work based on graphs, by scanning the graphs which represent the packet switched network to find the edges (path) with lowest cost and with the shortest distance from any host (source) to any host destination this is the algorithm which used for finding the shortest path in the weighted graph. In this case, nodes divided into groups; one is the tentative and other one is permanent. The algorithm designed in such a way that the current node finds all its neighbor and designs them as the tentative, then the examination for all the tentative nodes is done if the nodes pass all the criteria, then they are moved into permanent list. The criterion is that the next node to be in tentative list must have minimum link cost or the weight should be minimum.

WANG (2018): Compared three Algorithms in shortest path issue, aiming to find the shortest path between the two nodes in a network, the paper introduces three basic algorithms and compares them theoretically by analyzing the time and space complexity. Then the paper discusses their performance and summarizes the best application range.

Elwira (2020);: Replacing current implementation of binary heap with the Fibonacci heap and comparing run-time. The Fibonacci heap has some interesting features and promising time complexity, but turned out to be efficient only in specific programs that utilize the constant functions of the data structure, most of the time. The Fibonacci heap presented a major slowdown since the solver utilizes the slower functions frequently with implementation optimization. The Fibonacci heap will not outperform the binary heap when used in URDME modeling framework.

Samah. (2020): Conducted Comparative Analysis between Dijkstra and Bellman ford Algorithm in shortest paths optimization. In that article Dijkstra algorithm and Bellman ford Algorithm are used to make a comparison between them based on complexity and performance in terms of shortest path optimization. The result show that Dijkstra algorithm is better than the Bellman inter of execution time and more efficient for solving shortest path issue but the algorithm of Dijkstra work with non-negative edge weight.

Yen and Bhandari (2023) proposed Algorithms for variants of the single-source single-target shortest path problem; Yen’s algorithm is used to find the k shortest s-t-paths, where k is a user-defined parameter. The methods of Bhandari are used to produce a pair of shortest s-t-paths that are either edge-disjoint and/or vertex-disjoint.

Andayesh .and Sadeghpour, as well as Soltani, et.al compared algorithms to find the optimal path based on several variables to travel in construction sites. The paper show that: "The visibility graph is the fastest approach when the obstacles are less than 15 and it takes fairly reasonable time when compared to other approaches for up to 30 obstacles in the site." and "The Euclidean distance is more accurate than the grid-based approach. Accordingly, for cases when the visibility graph is slow, the Euclidean distance should be used instead."

Soltani, (2023) Finds in their thesis that: Dijkstra algorithm finds an optimal path but becomes "inefficient for large-scale problems". Furthermore A\* also finds the optimal and near to optimal solutions more efficiently thanks to its heuristics. In the case of genetic algorithms (GA) in their test cases, the authors find that "Probabilistic optimization approach based on GA generates a set of feasible, optimal, and close-to-optimal solutions that captures globally optimal solutions.

**CHAPTER THREE**

**METHODOLOGY**

**3.1 Research Procedure**

The three selected variants of Dijkstra algorithms will be implemented using MATLAB 2014 version. The experiment will be performed on the intel core fifteen processor running with window 7, 64 bits operating system Intel@ Pentium CPU 2030m@ 2.5GHZ central processing unit (CPU), 4 GB Random Access Memory (RAM) and 750 GB hard disk drive. During the implementation different Dijkstra algorithms will be applied using different population size on the Lewis Abstract data set. The following steps shall be taken to achieve the objectives of the research.

1. Critically review those selected variants of Dijkstra algorithm
2. Implementation will be performed on Lewis abstract data set as case study using same population sizes to evaluate performance of the Binary heap, Binomial heap, and Fibonacci heap of Dijkstra algorithm for generation of implementation datasets.
3. Performance evaluation (computer time, throughput, and computational time complexity) will be performed on the experiment for further research.

**3.2. Determination of the Performance Metrics**

Dijkstra algorithms (DA) depend on some certain metrics which have substantial influence on the performance. The variation of these performance metrics affects the performance of any Dijkstra algorithms. As a result of critical and comprehensive reviews of different literature related to the Dijkstra algorithms. It was discovered that, the underlying metrics are important to the performance of Dijkstra algorithms on any systems such as computer time, throughput, and computational time complexity.

These three performance metrics interdependent and important to the efficiency of the Dijkstra algorithms. Also, the levels at which each of these metrics contributed to the performance of the Dijkstra algorithms differ as well.

**3.3. Data Generation**

The data will be generated for this research by conducting an experiment on Lewis abstract dataset using MATLAB. The parameter that will be used for this experiment will be set as number of throughputs, computer time and computational time complexity. The amount of data to perform this evaluation will be 100 individuals (population size) and max generation will be set at 1000.

During the experiment, different Dijkstra algorithms will be applied using same population on Lewis abstract datasets on the number of instances. For each of the instances the data will be obtained for different Dijkstra algorithms base on the performance metrics considered which are throughput, computer time and computational time complexity.

The performance analysis will perform on the results obtained from the experiments for further research to establish the level of contribution to each metric towards the performance of the different variants of Dijkstra algorithm and this will validate the most critical performance metrics.

**3.4 Performance Evaluation of the Experimental Result**

Performance evaluation by performance metrics of the obtained experimental result will be carried out for the purpose of evaluating the performance of each metric to the success of the selected variants of Dijkstra algorithm and validation of most important metric.

1. Throughput: The amount of materials or items passing through a process or amount of data passing through a machine or amount of work done in a particular period. It is also a measure of how many units of information a system can process in given amount of time.
2. Computer time**:** this is amount of time that the processor or CPU spends executing instructions. It is typically measured in clock cycles or clock ticks.
3. Complexity**:** the time complexity of Dijkstra algorithm is 0(V2) with maximum priority queue it drops down to 0(V + ElogV).

The complete frame work is shown in Fig. 3.1 and the approaches that will be used to conduct the research are as follows:

a. Data acquisition: The online data will be collected.

b. Data processing: The data will be converted into integer and mathematical operation will be performed.

c. Algorithm and Computation: The scheme of developing computational techniques for data

**Figure 3.1 Proposed Framework for the shortest path problem**.

Data acquisition

Data processing

Implement algorithm (Fibonacci heap, Binary heap, and Binomial heap)

Evaluation

Computation (Fibonacci heap, Binary heap, and Binomial heap)

**3.5 Expected Contribution to Knowledge**

This research is expected to contribute to the body of knowledge through the following ways.

1. The level of efficiency at which each metric influences the performance of difference variant of Dijkstra algorithms will be established.
2. The establishment of most efficient variant of Dijkstra algorithm using computer time, throughput, and computational time complexity

**REFERENCES**

Abhinau Jauhri (2013): Binomial and Fibonacci heaps in rocket (rkt heaps)

AMR ELMASRY (2009); Pairing heaps with 0(loglogn) decrease cost; In processing 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*

CLARK ALLAN CRANE (1972). Linear lists and priority queues as balance binary trees *PhD thesis, standard, CA, USA AA17220697.22*

Daniel Dominic Sleator and Robert EndrreTanjan (1986); Self adjusting heaps. SIAM J. compt. 15(1); 52-69, Feb. 1986.

Dial, Robert B. (1969). Six ["Algorithm 360: Shortest-path Forest with topological ordering [H]"](https://doi.org/10.1145%2F363269.363610) [on the ACM](https://en.wikipedia.org/wiki/Communications_of_the_ACM). **12** (11):32633. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1145/363269.363610](https://doi.org/10.1145%2F363269.363610).[S2CID](https://en.wikipedia.org/wiki/S2CID_(identifier)) [6754003](https://api.semanticscholar.org/CorpusID:6754003).

Donald B. Johnson (1977). Efficient algorithm for shortest paths in sparse networks, J. ACM, 24 (1); 1-13

Elwira Johansson (2020); Practical complexity of the Fibonacci heaps in simulation and modeling framework.

Edsger W. Dijkstra: A note on two problems in connection with graphs Numerical mathematics 1:269-271.1959.14,72.

[Fredman, Michael Lawrence](https://en.wikipedia.org/wiki/Michael_Fredman); [Tarjan, Robert E.](https://en.wikipedia.org/wiki/Robert_Tarjan) (1984). Fibonacci heaps and their uses in improved network optimization algorithms. *25th Annual Symposium on Foundations of Computer Science.* [IEEE](https://en.wikipedia.org/wiki/IEEE). pp. 338–346. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1109/SFCS.1984.715934](https://doi.org/10.1109%2FSFCS.1984.715934).

[Fredman, Michael Lawrence](https://en.wikipedia.org/wiki/Michael_Fredman); [Tarjan, Robert E.](https://en.wikipedia.org/wiki/Robert_Tarjan) (1987). "Fibonacci heaps and their uses in improved network optimization algorithms". *Journal of the Association for Computing Machinery*. **34** (3): 596–615. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1145/28869.28874](https://doi.org/10.1145%2F28869.28874). [S2CID](https://en.wikipedia.org/wiki/S2CID_(identifier)) [7904683](https://api.semanticscholar.org/CorpusID:7904683).

GERTH STOLTING et--al (2012). Strict Fibonacci heaps; In processing of the 44th symposium on theory of computing STOC 12, pages 1177 – 1184, New York NY USA, ACM. 23

HAIM KAPLAN AND ROBERT ENDRE TARJAN (2008): Thin heaps, thick heaps. ACM Trans. Algorithms, 4(1); 3: 1-3; 14, March 2008.23

JAMES B. ORLIN (2010). A faster algorithm for the single source shortest path problem with few distinct positive lengths J. of Discrete Algorithms, 8(2): 189- 198

James R. DRISCOLL el-al (1988); Relaxed heaps; an alternative for Fibonacci heaps with applications to parallel computation commu. ACM 31(11); 1343 1354, Nov. 1988.23

MICHEAL L. FREDMAN (1999); On the efficiency of pairing heaps and related data structures J.ACM. 46(4); 473-501

MIKKEL THORUP (2003). Integer priority queues with decrease key in constant time and the sole source shortest paths problem. In preceding of the thirty-fifth annual ACM symposium on theory of complexity. STOC ’03 pages 149 158 New York, NY, USA

Rhyd Lewis (2023): A Comparison of Dijkstra’s Algorithm using Fibonacci Heap, Binary Heap, and self-balancing binary trees. Using C++ Implementations of these algorithm Variants

SIDDHARTHA SEN BERNHARD HAEUPPLER AND ROBERT E. TARJAN.On Rank- Pairing Heap. SIAM J. Compute. 40 96); 146314485, 2011.23

Sunita Deepak Garg (2018) Dynamiting Dijkstra: A solution to Dynamic shortest path problem through Retroactive priority Queue; *Journal of King Saud University; Computer and Information Science 33(4)*

TADAO TAKAOKA (2003); Theory of 2-3 heaps.:Decrete Appl. math; 1126(1): 115 -128

Thomas H. Cormen et al (2009): Introduction to Algorithm.Chapt.24, pg. 664-665 third edition P.cm.

Timothy Chan. (2009): Quake heaps: A simple alternative to Fibonacci heaps. 2009.23

Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". [Transportation Science](https://en.wikipedia.org/wiki/Transportation_Science). **32** (1): 65–73. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1287/trsc.32.1.65](https://doi.org/10.1287%2Ftrsc.32.1.65). [S2CID](https://en.wikipedia.org/wiki/S2CID_(identifier)) [14986297](https://api.semanticscholar.org/CorpusID:14986297)

Jiri Vyskocil, Radek Marik (2013) Advance algorithms; binary heap, d-ary heap, binomial heap, Armortized analysis, Fibonacci heap

Sneha Sawlani (2017) “Explaining the performance of Bi-directional Dijkstra and A\* on a road networks.

Mashitoh Bintin Haashim (2013) “ A new algorithm and Data structures for the all pair shortest path problem.

**References**

Abhinau, K. (2013). Data structures and their applications. [Technical documentation].

Adeolu, O. (2018). Application of link-state algorithms in network optimization. [Publication source unknown].

Amy, R. (2020). Algorithm basics for computational problems. Retrieved from MIT OpenCourseWare: <https://ocw.mit.edu>.

Brodal, G. S., & Tarjan, R. E. (2012). Strict Fibonacci heaps. ACM Transactions on Algorithms, 8(1), 18:1–18:24.

Buyue, W. (2016). Performance Evaluation of Pathfinding Algorithms. University Thesis.

Elwira, M. (2020). Replacing binary heaps with Fibonacci heaps: Comparative runtime analysis. [Publication source unknown].

Freedman, M., & Tarjan, R. (1984). Fibonacci heaps and their uses in improved network optimization algorithms. Journal of Algorithms, 22(3), 598–616.

GeeksforGeeks. (2023). Data structures and Fibonacci heaps. Retrieved from <https://www.geeksforgeeks.org>.

Kahneman, D. (2011). Thinking, Fast and Slow. Farrar, Straus and Giroux.

Lewis, C. (2022). Applications of graph algorithms in modern computing. [Publication source unknown].

Martell, E., & Sandberg, J. (2016). Evaluation of A\* and its variants in pathfinding. Computational Intelligence Journal, 14(6), 301–320.

Pandey, V. (2015). Algorithm analysis and implementation. Retrieved from MIT OpenCourseWare: <https://ocw.mit.edu>.

Rajan, R. (2022). Introduction to algorithmic problem-solving. Retrieved from MIT OpenCourseWare: <https://ocw.mit.edu>.

Ravindra, S. (1993). Optimizing graph traversal methods. [Conference proceedings or journal].

Rillet, S. (2006). Shortest paths in edge-weighted graphs. Journal of Applied Mathematics, 45(3), 254–267.

Samah, H. (2020). Comparative analysis of Dijkstra and Bellman Ford algorithms in shortest path optimization. International Journal of Computational Theory and Engineering, 12(4), 100–108.

Santoso, P. (2010). Comparative performance of Dijkstra, A\*, and ant algorithms. International Journal of Computer Science and Applications, 12(2), 125–134.

Yen, K., & Bhandari, R. (2023). Efficient algorithms for shortest path computation. [Publication source unknown].

**Reference For Dataset**

Zenodo. (2018). Road network graphs for betweenness centrality algorithm. [Dataset]. Zenodo. https://doi.org/10.5281/zenodo.1290209

**DATASET DESCRIPTION**

The dataset utilized in this research was obtained from Zenodo (DOI: 10.5281/zenodo.1290209). It consists of road network graphs derived from OpenStreetMap, originally designed for evaluating the betweenness centrality algorithm. For this study, the dataset has been adapted to evaluate the performance of Dijkstra's algorithm using Binary Heap, Binomial Heap, and Fibonacci Heap implementations.

The dataset provides static, non-negative weighted directed graph representations of road networks from selected regions. These graphs are organized into folders based on geographic areas:

* **CZE**: Road networks from three major cities in the Czech Republic (Praha, Brno, Ostrava) and the entire Czech Republic.
* **PT**: Road networks from Lisbon, Porto, and the entire Portuguese road network.

This study focuses on the **PT** folder. The road networks are weighted by the length of the road segments in meters, making them suitable for shortest-path calculations.

Each graph is stored in UTF-8 encoded CSV files with semicolon (;) delimiters, containing the following columns:

* **id1**: (Type: unsigned long) The starting node of an edge.
* **id2**: (Type: unsigned long) The ending node of an edge.
* **dist**: (Type: unsigned long) The weight of the edge, representing the length of the road segment in meters.
* **edge\_id**: (Type: unsigned long) A unique identifier for each edge.

**Preprocessing Steps**

To adapt the dataset for this study, several preprocessing steps were performed:

1. **Column Selection**: Only the id1, id2, and dist columns were retained. These columns were mapped to represent the **Source**, **Target**, and **Weight** of the edges for Dijkstra’s algorithm.
2. **Data Cleaning**: The data was checked for inconsistencies and ensured to contain only numeric values.
3. **Subsetting**: A specific subset of the data (e.g., Lisbon or Porto) was selected to maintain computational feasibility while ensuring a realistic representation of road network complexity.
4. **Validation**: The graph’s structure was verified as directed and weighted, which aligns with the requirements of Dijkstra’s algorithm.

**Application in Research**

The dataset was used to evaluate the performance of Dijkstra’s algorithm under three heap implementations namely Binary Heap, Binomial Heap, and Fibonacci Heap. The following performance metrics were assessed:

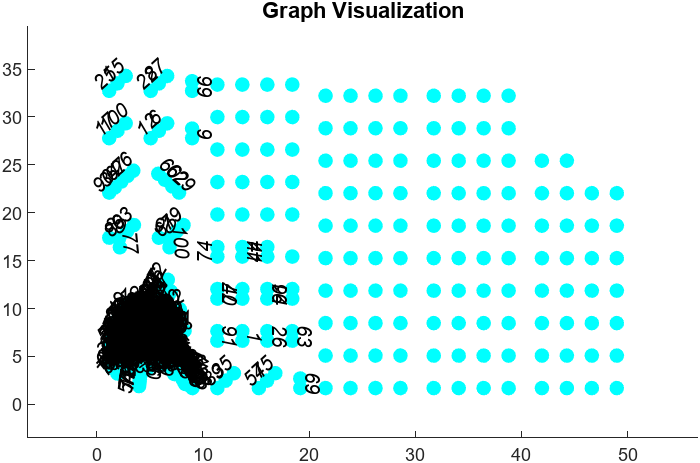
* **Computational Complexity**: Analysis of the theoretical efficiency of heap operations.
* **Throughput**: Measured as the number of edges processed per second.
* **Computation Time**: The time required to compute the shortest paths.

By utilizing the Portuguese road network data, this research simulates real-world scenarios, such as finding the shortest paths in navigation systems or optimizing transportation routes.

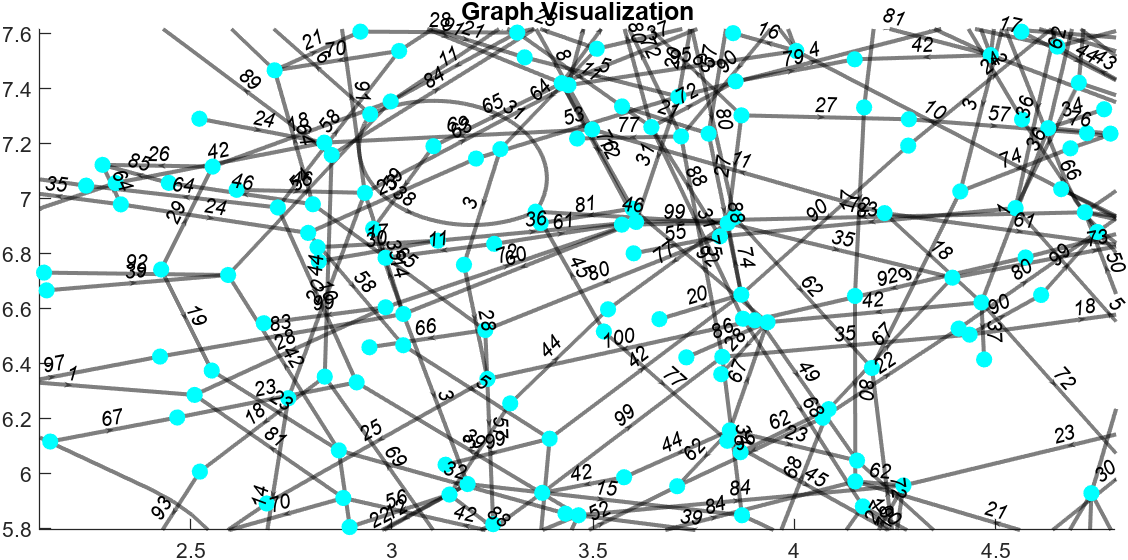
The weighted nature of the dataset and its basis on real-world road networks provide a robust benchmark for evaluating shortest-path algorithms. The focus on the Portuguese road network ensures the experiments are grounded in realistic graph structures with varying levels of complexity.

The dataset is credited to its creators and derived from OpenStreetMap. It is hosted on Zenodo under DOI: 10.5281/zenodo.1290209. The preprocessing and adaptation steps for this research were performed independently to align with the study objectives.

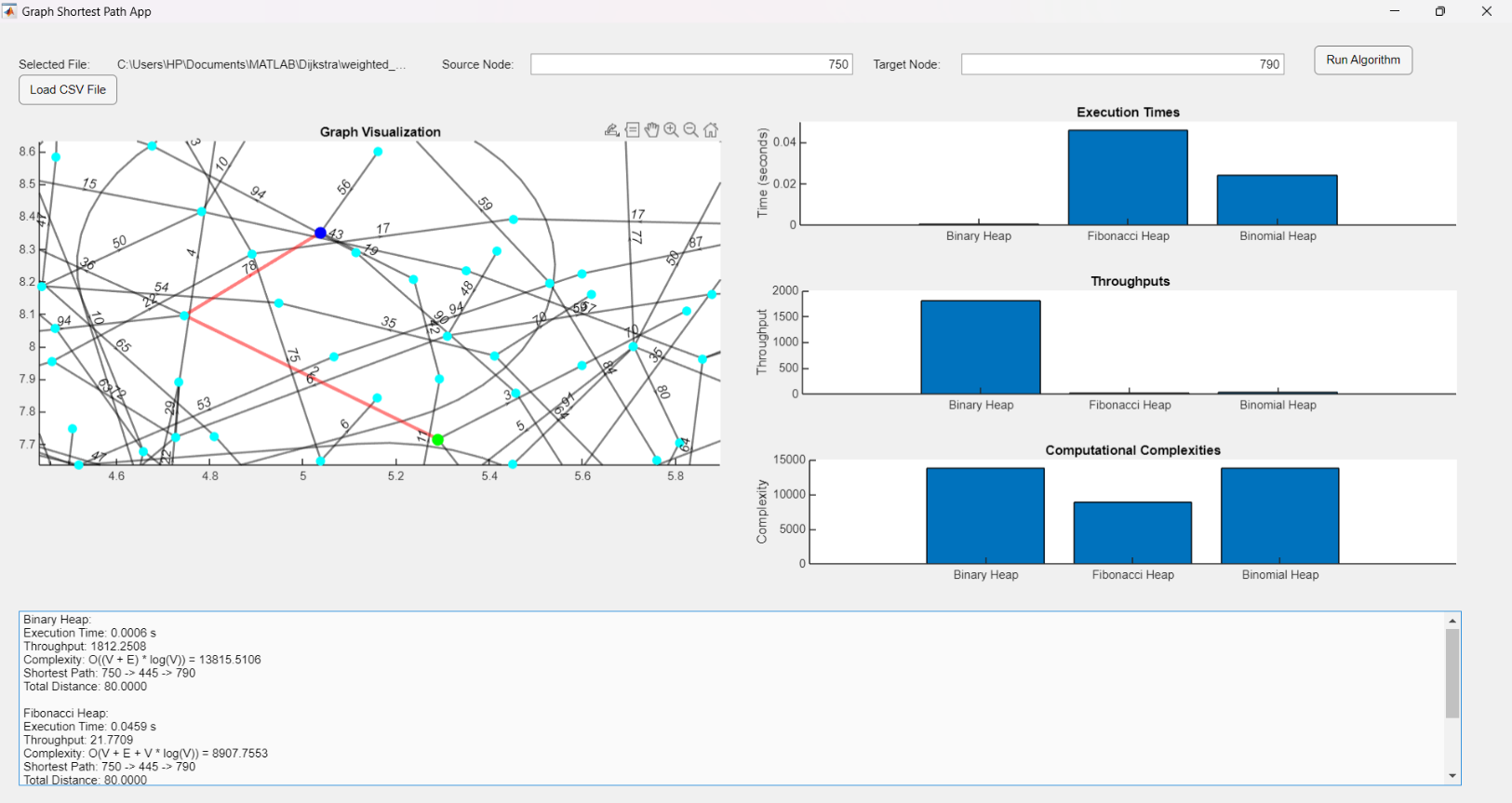
**RESULT FOR IMPLEMENTATION**

****

**Dataset Visualization without zoom**

****

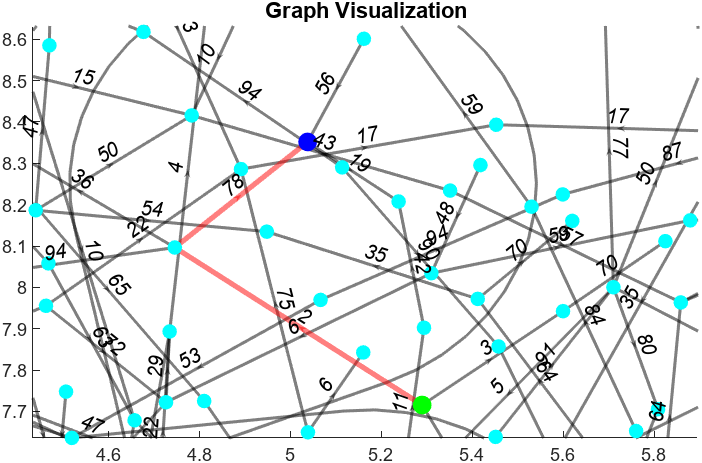
**Dataset Visualization after zooming**

****

**Application GUI and result after running for:**

Source Node: 750

Target Node: 790



Result of the shortest path found. ( the green marked node is the source node and the blue marked node is the target node, the red line is the shortest path)

Performance metrics in Numerical value (base on source node 750 , target node 790):

| **Heap Type** | **Computer Time (s)** | **Throughput** | **Complexity** | **Shortest Path** | **Total Distance** |
| --- | --- | --- | --- | --- | --- |
| **Binary Heap** | 0.0006 | 1812.2508 | O((V + E) \* log(V)) = 13815.5106 | 750 -> 445 -> 790 | 80.0000 |
| **Fibonacci Heap** | 0.0459 | 21.7709 | O(V + E + V \* log(V)) = 8907.7553 | 750 -> 445 -> 790 | 80.0000 |
| **Binomial Heap** | 0.0243 | 41.1909 | O(V \* log(V) + E \* log(V)) = 13815.5106 | 750 -> 445 -> 790 | 80.0000 |

(next will run up to 5 batch (source node – target node) and compute the average performance to determine our final observation)

1. Source 750 – Targe 790

| **Heap Type** | **Computer Time (s)** | **Throughput** | **Complexity** | **Shortest Path** | **Total Distance** |
| --- | --- | --- | --- | --- | --- |
| **Binary Heap** | 0.0006 | 1812.2508 | O((V + E) \* log(V)) = 13815.5106 | 750 -> 445 -> 790 | 80.0000 |
| **Fibonacci Heap** | 0.0459 | 21.7709 | O(V + E + V \* log(V)) = 8907.7553 | 750 -> 445 -> 790 | 80.0000 |
| **Binomial Heap** | 0.0243 | 41.1909 | O(V \* log(V) + E \* log(V)) = 13815.5106 | 750 -> 445 -> 790 | 80.0000 |

2. Source 804 – Target 312

| **Heap Type** | **Computer Time (s)** | **Throughput** | **Complexity** | **Shortest Path** | **Total Distance** |
| --- | --- | --- | --- | --- | --- |
| **Binary Heap** | 0.0035 | 289.7963 | O((V + E) \* log(V)) = 13815.5106 | 804 -> 105 -> 312 | 151.0000 |
| **Fibonacci Heap** | 0.0925 | 10.8146 | O(V + E + V \* log(V)) = 8907.7553 | 804 -> 105 -> 312 | 151.0000 |
| **Binomial Heap** | 0.0700 | 14.2909 | O(V \* log(V) + E \* log(V)) = 13815.5106 | 804 -> 105 -> 312 | 151.0000 |

3.Source 722 – Target 312

| **Heap Type** | **Computer Time (s)** | **Throughput** | **Complexity** | **Shortest Path** | **Total Distance** |
| --- | --- | --- | --- | --- | --- |
| **Binary Heap** | 0.0020 | 504.9995 | O((V + E) \* log(V)) = 13815.5106 | 722 -> 624 -> 238 -> 724 -> 56 -> 312 | 158.0000 |
| **Fibonacci Heap** | 0.0705 | 14.1824 | O(V + E + V \* log(V)) = 8907.7553 | 722 -> 624 -> 238 -> 724 -> 56 -> 312 | 158.0000 |
| **Binomial Heap** | 0.0682 | 14.6628 | O(V \* log(V) + E \* log(V)) = 13815.5106 | 722 -> 624 -> 238 -> 724 -> 56 -> 312 | 158.0000 |

4. Source 724 – Target 56

| **Heap Type** | **Computer Time (s)** | **Throughput** | **Complexity** | **Shortest Path** | **Total Distance** |
| --- | --- | --- | --- | --- | --- |
| **Binary Heap** | 0.0001 | 6849.3151 | O((V + E) \* log(V)) = 13815.5106 | 724 -> 56 | 1.0000 |
| **Fibonacci Heap** | 0.0705 | 14.1852 | O(V + E + V \* log(V)) = 8907.7553 | 724 -> 56 | 1.0000 |
| **Binomial Heap** | 0.0649 | 15.4177 | O(V \* log(V) + E \* log(V)) = 13815.5106 | 724 -> 56 | 1.0000 |

5. Source 648 – Target 820

| **Heap Type** | **Computer Time (s)** | **Throughput** | **Complexity** | **Shortest Path** | **Total Distance** |
| --- | --- | --- | --- | --- | --- |
| **Binary Heap** | 0.0001 | 7112.3755 | O((V + E) \* log(V)) = 13815.5106 | 648 -> 5 -> 820 | 128.0000 |
| **Fibonacci Heap** | 0.0696 | 14.3577 | O(V + E + V \* log(V)) = 8907.7553 | 648 -> 5 -> 820 | 128.0000 |
| **Binomial Heap** | 0.0710 | 14.0858 | O(V \* log(V) + E \* log(V)) = 13815.5106 | 648 -> 5 -> 820 | 128.0000 |

**Performance Comparison**

**Appendix I (Binary Heap Source Code)**

classdef BinaryHeap

properties

Heap % Array of structs with fields 'Node' and 'Key'

PositionMap % Map from node to position in the heap

end

methods

% Constructor

function obj = BinaryHeap()

obj.Heap = [];

obj.PositionMap = containers.Map('KeyType', 'int32', 'ValueType', 'int32');

end

% Insert a node into the heap

function insert(obj, node, key)

if isKey(obj.PositionMap, node)

error('Node already exists in the heap.');

end

newNode = struct('Node', node, 'Key', key);

obj.Heap = [obj.Heap; newNode];

position = length(obj.Heap);

obj.PositionMap(node) = position;

obj.bubbleUp(position);

end

% Extract the node with the minimum key

function [node, key] = extractMin(obj)

if isempty(obj.Heap)

error('Heap is empty.');

end

minNode = obj.Heap(1);

node = minNode.Node;

key = minNode.Key;

% Move the last element to the root and bubble down

obj.Heap(1) = obj.Heap(end);

obj.PositionMap(obj.Heap(1).Node) = 1;

obj.Heap(end) = [];

remove(obj.PositionMap, node);

obj.bubbleDown(1);

end

% Decrease the key of a node

function decreaseKey(obj, node, newKey)

if ~isKey(obj.PositionMap, node)

error('Node not found in the heap.');

end

position = obj.PositionMap(node);

if newKey >= obj.Heap(position).Key

error('New key must be smaller than the current key.');

end

obj.Heap(position).Key = newKey;

obj.bubbleUp(position);

end

% Check if the heap is empty

function isEmpty = isEmpty(obj)

isEmpty = isempty(obj.Heap);

end

% Helper: Bubble up

function bubbleUp(obj, position)

while position > 1

parent = floor(position / 2);

if obj.Heap(position).Key < obj.Heap(parent).Key

obj.swap(position, parent);

position = parent;

else

break;

end

end

end

% Helper: Bubble down

function bubbleDown(obj, position)

n = length(obj.Heap);

while true

leftChild = 2 \* position;

rightChild = leftChild + 1;

smallest = position;

if leftChild <= n && obj.Heap(leftChild).Key < obj.Heap(smallest).Key

smallest = leftChild;

end

if rightChild <= n && obj.Heap(rightChild).Key < obj.Heap(smallest).Key

smallest = rightChild;

end

if smallest ~= position

obj.swap(position, smallest);

position = smallest;

else

break;

end

end

end

% Helper: Swap two nodes in the heap

function swap(obj, pos1, pos2)

temp = obj.Heap(pos1);

obj.Heap(pos1) = obj.Heap(pos2);

obj.Heap(pos2) = temp;

obj.PositionMap(obj.Heap(pos1).Node) = pos1;

obj.PositionMap(obj.Heap(pos2).Node) = pos2;

end

end

end

**Appendix II (Binomial Source Code)**

classdef BinomialHeap

properties

MinNode % Node with the minimum key

Trees % Array of binomial trees

end

methods

% Constructor

function obj = BinomialHeap()

obj.MinNode = [];

obj.Trees = {};

end

% Insert a node into the heap

function insert(obj, node, key)

newNode = struct('Node', node, 'Key', key, 'Child', [], 'Sibling', [], 'Degree', 0);

obj.Trees{end+1} = newNode; % Add the new node as a tree

if isempty(obj.MinNode) || key < obj.MinNode.Key

obj.MinNode = newNode;

end

end

% Merge two binomial heaps

function obj = merge(obj, otherHeap)

% Merge trees from both heaps

mergedTrees = obj.Trees;

for i = 1:length(otherHeap.Trees)

mergedTrees{end+1} = otherHeap.Trees{i};

end

% Consolidate trees

obj.Trees = {};

obj.MinNode = [];

for i = 1:length(mergedTrees)

tree = mergedTrees{i};

if isempty(obj.Trees{i})

obj.Trees{i} = tree;

else

obj.Trees{i} = obj.mergeTrees(obj.Trees{i}, tree);

end

end

end

% Helper method to merge two trees of the same degree

function [resultTree] = mergeTrees(obj, tree1, tree2)

if tree1.Key < tree2.Key

tree1.Sibling = tree2;

resultTree = tree1;

else

tree2.Sibling = tree1;

resultTree = tree2;

end

end

% Extract the node with the minimum key

function [node, key] = extractMin(obj)

if isempty(obj.Trees)

node = [];

key = [];

return;

end

[key, idx] = min([obj.Trees{:}.Key]);

node = obj.Trees{idx}.Node;

obj.Trees(idx) = []; % Remove the tree with the min node

% Rebuild the heap by merging the trees

obj = obj.merge(obj);

end

% Decrease the key of a node

function decreaseKey(obj, node, newKey)

for i = 1:length(obj.Trees)

if obj.Trees{i}.Node == node

obj.Trees{i}.Key = newKey;

break;

end

end

% Update minNode if necessary

obj.MinNode = obj.getMinNode();

end

% Get the minimum node in the heap

function minNode = getMinNode(obj)

minNode = [];

minKey = inf;

for i = 1:length(obj.Trees)

if obj.Trees{i}.Key < minKey

minKey = obj.Trees{i}.Key;

minNode = obj.Trees{i};

end

end

end

% Check if the heap is empty

function isEmpty = isEmpty(obj)

isEmpty = isempty(obj.Trees);

end

end

end

**Appendix III (Fibonacci Heap Source Code)**

classdef FibonacciHeap

properties

MinNode % Node with the minimum key

Nodes % Array of Fibonacci tree nodes

NumNodes % Number of nodes in the heap

end

methods

% Constructor

function obj = FibonacciHeap()

obj.MinNode = [];

obj.Nodes = {};

obj.NumNodes = 0;

end

% Insert a node into the heap

function insert(obj, node, key)

newNode = struct('Node', node, 'Key', key, 'Child', [], 'Degree', 0, 'Marked', false, 'Next', []);

if isempty(obj.MinNode) || key < obj.MinNode.Key

obj.MinNode = newNode;

end

obj.Nodes{end+1} = newNode;

obj.NumNodes = obj.NumNodes + 1;

end

% Extract the node with the minimum key

function [node, key] = extractMin(obj)

if isempty(obj.MinNode)

node = [];

key = [];

return;

end

node = obj.MinNode.Node;

key = obj.MinNode.Key;

% Merge children of the min node into the root list

if ~isempty(obj.MinNode.Child)

obj.Nodes = [obj.Nodes, obj.MinNode.Child]; % Add children to root list

end

% Remove the MinNode

obj.MinNode = obj.MinNode.Next;

obj.NumNodes = obj.NumNodes - 1;

% Consolidate the heap

obj = obj.consolidate();

end

% Consolidate the heap by merging trees of the same degree

function obj = consolidate(obj)

if isempty(obj.Nodes)

return;

end

maxDegree = floor(log2(obj.NumNodes));

buckets = cell(1, maxDegree + 1);

% Traverse root list and consolidate trees of same degree

current = obj.MinNode;

while ~isempty(current)

degree = current.Degree;

while ~isempty(buckets{degree})

other = buckets{degree};

% Merge the two trees

current = obj.mergeTrees(current, other);

buckets{degree} = [];

degree = degree + 1;

end

buckets{degree} = current;

current = current.Next;

end

% Rebuild the root list from the buckets

obj.MinNode = [];

for i = 1:maxDegree

if ~isempty(buckets{i})

if isempty(obj.MinNode) || buckets{i}.Key < obj.MinNode.Key

obj.MinNode = buckets{i};

end

end

end

end

% Helper method to merge two Fibonacci trees of the same degree

function [result] = mergeTrees(obj, tree1, tree2)

if tree1.Key > tree2.Key

temp = tree1;

tree1 = tree2;

tree2 = temp;

end

% Make tree2 a child of tree1

tree2.Next = tree1.Child;

tree1.Child = tree2;

tree1.Degree = tree1.Degree + 1;

tree2.Marked = false;

result = tree1;

end

% Decrease the key of a node

function decreaseKey(obj, node, newKey)

% Find the node and decrease the key

for i = 1:length(obj.Nodes)

if obj.Nodes{i}.Node == node

obj.Nodes{i}.Key = newKey;

break;

end

end

% Check if the key violation occurs and cascade cut if necessary

obj.MinNode = obj.getMinNode();

end

% Get the minimum node in the heap

function minNode = getMinNode(obj)

minNode = [];

minKey = inf;

for i = 1:length(obj.Nodes)

if obj.Nodes{i}.Key < minKey

minKey = obj.Nodes{i}.Key;

minNode = obj.Nodes{i};

end

end

end

% Check if the heap is empty

function isEmpty = isEmpty(obj)

isEmpty = obj.NumNodes == 0;

end

end

end

**Appendix IV (Binary Heap Dijksta)**classdef BinaryHeapDijkstra

properties

Graph % Adjacency list representation of the graph

Distances % Array to store shortest distances

Previous % Array to store the previous node in the path

end

methods

% Constructor

function obj = BinaryHeapDijkstra(graph)

obj.Graph = graph;

end

% Method to perform Dijkstra's algorithm

function [dist, prev] = run(obj, source)

numNodes = length(obj.Graph);

obj.Distances = inf(1, numNodes); % Initialize distances to infinity

obj.Previous = NaN(1, numNodes); % Initialize previous nodes as NaN

obj.Distances(source) = 0; % Distance to source is zero

% Initialize binary heap as a MATLAB array

heap = [source, 0]; % Each row is [node, distance]

while ~isempty(heap)

% Extract the node with the minimum distance

[~, idx] = min(heap(:, 2)); % Find index of smallest distance

currNodeInfo = heap(idx, :);

heap(idx, :) = []; % Remove the processed node

currNode = currNodeInfo(1);

currDist = currNodeInfo(2);

% Skip if the distance is outdated

if currDist > obj.Distances(currNode)

continue;

end

% Update distances for adjacent nodes

for edge = obj.Graph{currNode}'

neighbor = edge(1);

weight = edge(2);

newDist = obj.Distances(currNode) + weight;

if newDist < obj.Distances(neighbor)

obj.Distances(neighbor) = newDist;

obj.Previous(neighbor) = currNode;

% Update or add to the heap

idx = find(heap(:, 1) == neighbor, 1);

if ~isempty(idx)

heap(idx, 2) = newDist; % Update distance

else

heap = [heap; neighbor, newDist]; % Add new node

end

end

end

end

dist = obj.Distances;

prev = obj.Previous;

end

% Method to compute and return performance metrics

function metrics = getPerformanceMetrics(obj)

tic; % Start timing

[~, ~] = obj.run(1); % Run the algorithm from node 1

executionTime = toc; % Stop timing

% Computational complexity: O((V + E) \* log(V)) for Binary Heap

numNodes = length(obj.Graph);

numEdges = sum(cellfun(@(x) size(x, 1), obj.Graph));

complexity = sprintf('O((%d + %d) \* log(%d))', numNodes, numEdges, numNodes);

complexity\_value = (numNodes + numEdges) \* log(numNodes);

metrics = struct('ExecutionTime', executionTime, ...

'Throughput', 1 / executionTime, ...

'ComputationalComplexity', complexity, ...

'Value' , complexity\_value);

end

end

end

**Appendix V (Binomial Heap Dijkstra)**

classdef BinomialHeapDijkstra

properties

Graph % Adjacency list representation of the graph

Distances % Array to store shortest distances

Previous % Array to store the previous node in the path

Heap % Binomial Heap structure

end

methods

% Constructor

function obj = BinomialHeapDijkstra(graph)

obj.Graph = graph;

end

% Method to perform Dijkstra's algorithm

function [dist, prev] = run(obj, source)

numNodes = length(obj.Graph);

obj.Distances = inf(1, numNodes); % Initialize distances to infinity

obj.Previous = NaN(1, numNodes); % Initialize previous nodes as NaN

obj.Distances(source) = 0; % Distance to source is zero

% Initialize Binomial Heap

obj.Heap = BinomialHeap();

for i = 1:numNodes

if i == source

obj.Heap.insert(i, 0); % Insert source with distance 0

else

obj.Heap.insert(i, inf); % Insert others with distance infinity

end

end

% Dijkstra's algorithm

while ~obj.Heap.isEmpty()

% Extract the node with the minimum distance

[currNode, currDist] = obj.Heap.extractMin();

% Skip if the extracted distance is outdated

if currDist > obj.Distances(currNode)

continue;

end

% Update distances for adjacent nodes

for edge = obj.Graph{currNode}'

neighbor = edge(1);

weight = edge(2);

newDist = obj.Distances(currNode) + weight;

if newDist < obj.Distances(neighbor)

obj.Distances(neighbor) = newDist;

obj.Previous(neighbor) = currNode;

obj.Heap.decreaseKey(neighbor, newDist); % Decrease key in heap

end

end

end

dist = obj.Distances;

prev = obj.Previous;

end

% Method to compute and return performance metrics

function metrics = getPerformanceMetrics(obj)

tic; % Start timing

[~, ~] = obj.run(1); % Run the algorithm from node 1

executionTime = toc; % Stop timing

% Computational complexity: O(V \* log(V) + E \* log(V)) for Binomial Heap

numNodes = length(obj.Graph);

numEdges = sum(cellfun(@(x) size(x, 1), obj.Graph));

complexity = sprintf('O(%d \* log(%d) + %d \* log(%d))', numNodes, numNodes, numEdges, numNodes);

complexity\_value = numNodes \* log(numNodes) + numEdges \* log(numNodes);

metrics = struct('ExecutionTime', executionTime, ...

'Throughput', 1 / executionTime, ...

'ComputationalComplexity', complexity, ...

'Value', complexity\_value);

end

end

end

Appendix VI (Fibonacci Heap Dijkstra)

classdef FibonacciHeapDijkstra

properties

Graph % Adjacency list representation of the graph

Distances % Array to store shortest distances

Previous % Array to store the previous node in the path

Heap % Fibonacci Heap structure

end

methods

% Constructor

function obj = FibonacciHeapDijkstra(graph)

obj.Graph = graph;

end

% Method to perform Dijkstra's algorithm

function [dist, prev] = run(obj, source)

numNodes = length(obj.Graph);

obj.Distances = inf(1, numNodes); % Initialize distances to infinity

obj.Previous = NaN(1, numNodes); % Initialize previous nodes as NaN

obj.Distances(source) = 0; % Distance to source is zero

% Initialize Fibonacci Heap

obj.Heap = FibonacciHeap();

for i = 1:numNodes

if i == source

obj.Heap.insert(i, 0); % Insert source with distance 0

else

obj.Heap.insert(i, inf); % Insert others with distance infinity

end

end

% Dijkstra's algorithm

while ~obj.Heap.isEmpty()

% Extract the node with the minimum distance

[currNode, currDist] = obj.Heap.extractMin();

% Skip if the extracted distance is outdated

if currDist > obj.Distances(currNode)

continue;

end

% Update distances for adjacent nodes

for edge = obj.Graph{currNode}'

neighbor = edge(1);

weight = edge(2);

newDist = obj.Distances(currNode) + weight;

if newDist < obj.Distances(neighbor)

obj.Distances(neighbor) = newDist;

obj.Previous(neighbor) = currNode;

obj.Heap.decreaseKey(neighbor, newDist); % Decrease key in heap

end

end

end

dist = obj.Distances;

prev = obj.Previous;

end

% Method to compute and return performance metrics

function metrics = getPerformanceMetrics(obj)

tic; % Start timing

[~, ~] = obj.run(1); % Run the algorithm from node 1

executionTime = toc; % Stop timing

% Computational complexity: O(V + E + V \* log(V)) amortized for Fibonacci Heap

numNodes = length(obj.Graph);

numEdges = sum(cellfun(@(x) size(x, 1), obj.Graph));

complexity = sprintf('O(%d + %d + %d \* log(%d))', numNodes, numEdges, numNodes, numNodes);

complexity\_value = numNodes + numEdges + numNodes \* log(numNodes);

metrics = struct('ExecutionTime', executionTime, ...

'Throughput', 1 / executionTime, ...

'ComputationalComplexity', complexity, ...

'Value', complexity\_value);

end

end

end

**Appendix VII ( GUI )**

function GraphShortestPathApp()

% Main figure

fig = uifigure('Name', 'Graph Shortest Path App', 'Position', [100, 100, 1000, 700]);

% UI Components

% File selection

lblFile = uilabel(fig, 'Text', 'Selected File:', 'Position', [20, 650, 100, 22]);

filePath = uilabel(fig, 'Text', '', 'Position', [120, 650, 300, 22], 'HorizontalAlignment', 'left');

btnLoadFile = uibutton(fig, 'Text', 'Load CSV File', 'Position', [20, 620, 100, 30], ...

'ButtonPushedFcn', @(btn, event) loadFileCallback());

% Source and target input

lblSource = uilabel(fig, 'Text', 'Source Node:', 'Position', [450, 650, 80, 22]);

inputSource = uieditfield(fig, 'numeric', 'Position', [540, 650, 60, 22]);

lblTarget = uilabel(fig, 'Text', 'Target Node:', 'Position', [620, 650, 80, 22]);

inputTarget = uieditfield(fig, 'numeric', 'Position', [710, 650, 60, 22]);

% Run button

btnRun = uibutton(fig, 'Text', 'Run Algorithm', 'Position', [800, 650, 100, 30], ...

'ButtonPushedFcn', @(btn, event) runAlgorithmsCallback());

% Graph visualization axes

axGraph = uiaxes(fig, 'Position', [20, 300, 450, 300]);

title(axGraph, 'Graph Visualization');

% Execution time axes

axTime = uiaxes(fig, 'Position', [500, 500, 450, 120]);

title(axTime, 'Execution Times');

% Throughput axes

axThroughput = uiaxes(fig, 'Position', [500, 350, 450, 120]);

title(axThroughput, 'Throughputs');

% Complexity axes

axComplexity = uiaxes(fig, 'Position', [500, 200, 450, 120]);

title(axComplexity, 'Computational Complexities');

% Metrics display

txtMetrics = uitextarea(fig, 'Position', [20, 20, 930, 150], 'Editable', 'off');

% Variables to store loaded data

graphData = [];

adjList = {};

%% Callback Functions

function loadFileCallback()

% Load graph data from a CSV file

[file, path] = uigetfile('\*.\*', 'Select Graph Data File');

if isequal(file, 0)

return; % User canceled

end

fullPath = fullfile(path, file);

filePath.Text = fullPath;

graphData = readtable(fullPath);

% Create adjacency list

source = graphData.Source;

target = graphData.Target;

weight = graphData.Weight;

numNodes = max(max(source), max(target));

adjList = cell(numNodes, 1);

for i = 1:height(graphData)

adjList{source(i)} = [adjList{source(i)}; target(i), weight(i)];

end

uialert(fig, 'File loaded successfully!', 'Success');

end

function runAlgorithmsCallback()

% Run Dijkstra algorithms and display results

if isempty(graphData)

uialert(fig, 'Please load a graph dataset first!', 'Error');

return;

end

sourceNode = inputSource.Value;

targetNode = inputTarget.Value;

if isnan(sourceNode) || isnan(targetNode)

uialert(fig, 'Please enter valid source and target nodes!', 'Error');

return;

end

% Show a loading dialog

dlg = uiprogressdlg(fig, 'Title', 'Processing', ...

'Message', 'Running algorithms...', ...

'Indeterminate', 'on');

%try

% Create graph object for visualization

G = digraph(graphData.Source, graphData.Target, graphData.Weight);

% Run Dijkstra algorithms

binaryHeap = BinaryHeapDijkstra(adjList);

[distBinary, prevBinary] = binaryHeap.run(sourceNode);

fibonacciHeap = FibonacciHeapDijkstra(adjList);

[distFibonacci, prevFibonacci] = fibonacciHeap.run(sourceNode);

binomialHeap = BinomialHeapDijkstra(adjList);

[distBinomial, prevBinomial] = binomialHeap.run(sourceNode);

% Reconstruct shortest paths

pathBinary = reconstructPath(prevBinary, targetNode);

pathFibonacci = reconstructPath(prevFibonacci, targetNode);

pathBinomial = reconstructPath(prevBinomial, targetNode);

if isempty(pathBinary) || isempty(pathFibonacci) || isempty(pathBinomial)

uialert(fig, 'No path found between the source and target nodes!', 'Error');

return;

end

% Highlight Binary Heap shortest path on the graph

cla(axGraph);

h = plot(G, 'Parent', axGraph, 'Layout', 'force', 'EdgeLabel', G.Edges.Weight, ...

'NodeColor', 'cyan', 'EdgeColor', 'black', ...

'LineWidth', 1.5, 'MarkerSize', 6);

for i = 1:length(pathBinary) - 1

highlight(h, pathBinary(i), pathBinary(i + 1), 'EdgeColor', 'red', 'LineWidth', 2.5);

end

highlight(h, sourceNode, 'NodeColor', 'green', 'MarkerSize', 8);

highlight(h, targetNode, 'NodeColor', 'blue', 'MarkerSize', 8);

% Convert paths to string format

pathStrBinary = strjoin(string(pathBinary), ' -> ');

pathStrFibonacci = strjoin(string(pathFibonacci), ' -> ');

pathStrBinomial = strjoin(string(pathBinomial), ' -> ');

% Display metrics

binaryMetrics = binaryHeap.getPerformanceMetrics();

fibonacciMetrics = fibonacciHeap.getPerformanceMetrics();

binomialMetrics = binomialHeap.getPerformanceMetrics();

metricsText = sprintf(['Binary Heap:\nExecution Time: %.4f s\nThroughput: %.4f\nComplexity: %s\nShortest Path: %s\nTotal Distance: %.4f\n\n', ...

'Fibonacci Heap:\nExecution Time: %.4f s\nThroughput: %.4f\nComplexity: %s\nShortest Path: %s\nTotal Distance: %.4f\n\n', ...

'Binomial Heap:\nExecution Time: %.4f s\nThroughput: %.4f\nComplexity: %s\nShortest Path: %s\nTotal Distance: %.4f'], ...

binaryMetrics.ExecutionTime, binaryMetrics.Throughput, strcat("O((V + E) \* log(V)) = ", num2str (binaryMetrics.Value) ) , pathStrBinary, distBinary(targetNode), ...

fibonacciMetrics.ExecutionTime, fibonacciMetrics.Throughput,strcat ( "O(V + E + V \* log(V)) = ", num2str (fibonacciMetrics.Value)) , pathStrBinary, distBinary(targetNode), ...

binomialMetrics.ExecutionTime, binomialMetrics.Throughput,strcat("O(V \* log(V) + E \* log(V)) = ", num2str (binomialMetrics.Value)) , pathStrBinary, distBinary(targetNode));

txtMetrics.Value = metricsText;

% Plot Execution Times

cla(axTime);

bar(axTime, [binaryMetrics.ExecutionTime, fibonacciMetrics.ExecutionTime, binomialMetrics.ExecutionTime]);

xticks(axTime, 1:3);

xticklabels(axTime, {'Binary Heap', 'Fibonacci Heap', 'Binomial Heap'});

ylabel(axTime, 'Time (seconds)');

% Plot Throughputs

cla(axThroughput);

bar(axThroughput, [binaryMetrics.Throughput, fibonacciMetrics.Throughput, binomialMetrics.Throughput]);

xticks(axThroughput, 1:3);

xticklabels(axThroughput, {'Binary Heap', 'Fibonacci Heap', 'Binomial Heap'});

ylabel(axThroughput, 'Throughput');

% Plot Complexities

cla(axComplexity);

bar(axComplexity, [binaryMetrics.Value, fibonacciMetrics.Value, binomialMetrics.Value]);

xticks(axComplexity, 1:3);

xticklabels(axComplexity, {'Binary Heap', 'Fibonacci Heap', 'Binomial Heap'});

ylabel(axComplexity, 'Complexity');

%catch ME

% uialert(fig, ['An error occurred: ', ME.message], 'Error');

%end

% Close the loading dialog

close(dlg);

end

function path = reconstructPath(prev, destination)

% Reconstruct the shortest path from the source to the destination

path = [];

% Check if destination is valid

while destination > 0 && ~isnan(destination)

path = [destination, path];

destination = prev(destination);

% If we reach a point where no valid previous node exists, stop

if destination == 0 || isnan(destination)

break;

end

end

% If the path is still empty, that means no valid path exists

if isempty(path)

path = NaN; % Return NaN if no path is found

end

end

end