CCP 6214 Algorithm Design and Analysis

Assignment Report

TC4L

T15L

Group 7

Num	Student ID	Student Name	Task Description	Percentage %
1	1211101007	Aisyah Binti Ahmad Kassim	Q1: Dataset 1 and 2	25
2	1211200107	Afiezar Ilyaz bin Alfie Iskandar	Q2: Heap and Selection Sort	25
3	1221303660	Wong Wai Yee	Q3: Dijkstra's and Kruskal	25
4	1211202025	Abdullah Bin Kamaruddin (Group Leader)	Q4: 0/1 Knapsack	25

1. Dataset generation for Dataset 1 and Dataset 2

a) Algorithms

i. Dataset 1

Problem Statement:

Write an algorithm to generate 6 sets of data with random numeric seeding. Use group leader's ID as the random seed reference to generate dataset of size(s) as follows:

Set 1: 100, Set 2: 1000, Set 3: 10000, Set 4: 100000, Set 5: 500000, Set 6: 1000000

Algorithm Description:

1. Extract Unique Digits from a Number:
    - This function takes a number, extracts each digit, and inserts it into an unordered set to ensure all digits are unique.
    - Finally, it converts the set to a vector and returns it.

2. Generate a Dataset of Random 3-Digit Numbers:
    - This function takes a seed for the random number generator, the size of the dataset, and a vector of allowed digits.
    - It initializes the random number generator with the given seed and reserves space for the dataset.
    - For each number to be generated, it constructs a 3-digit number by randomly selecting digits from the allowed_digits vector.
    - It then adds the generated number to the dataset.

3. Save a Dataset to a File:
    - This function opens a file for writing, checks if the file is successfully opened, and then writes each number from the dataset to the file.
    - It prints a message upon successful completion or an error message if the file cannot be opened.

4. Generate and Save Multiple Datasets:
    - This function extracts unique digits from the group leader's ID to form the allowed_digits vector.
    - It then iterates through the predefined dataset sizes, generates datasets with increasing sizes, and saves them to files named dataset_1.txt, dataset_2.txt, etc.
    - Each dataset is generated using a different seed (id_leader + i).

ii. Dataset 2

Problem Statement:

Write an algorithm to generate a dataset for 20 locations of stars (vertices) with 54 routes (edges) that connect between the stars. Each star needs to at least connect to 3 other stars. Use the sum of other group members' ID number to generate data consisting of values for star name, x-coordinate, y-coordinate, z-coordinate, weight and profit. The weight and profit refer to the points to be collected from conquering the star.

Algorithm Description:

1. Extract Unique Digits from a Number:
    - This function extracts unique digits from a given number. This is essential for ensuring the randomness in the generation process is based on a specific set of digits.
    - Converts the number into its individual digits, stores them in an unordered set to ensure uniqueness, then returns the unique digits as a vector.

2. Generate Unique Random Numbers:
    - Uses a custom random function to select digits from a set of allowed digits and constructs a number with a specified number of digits.

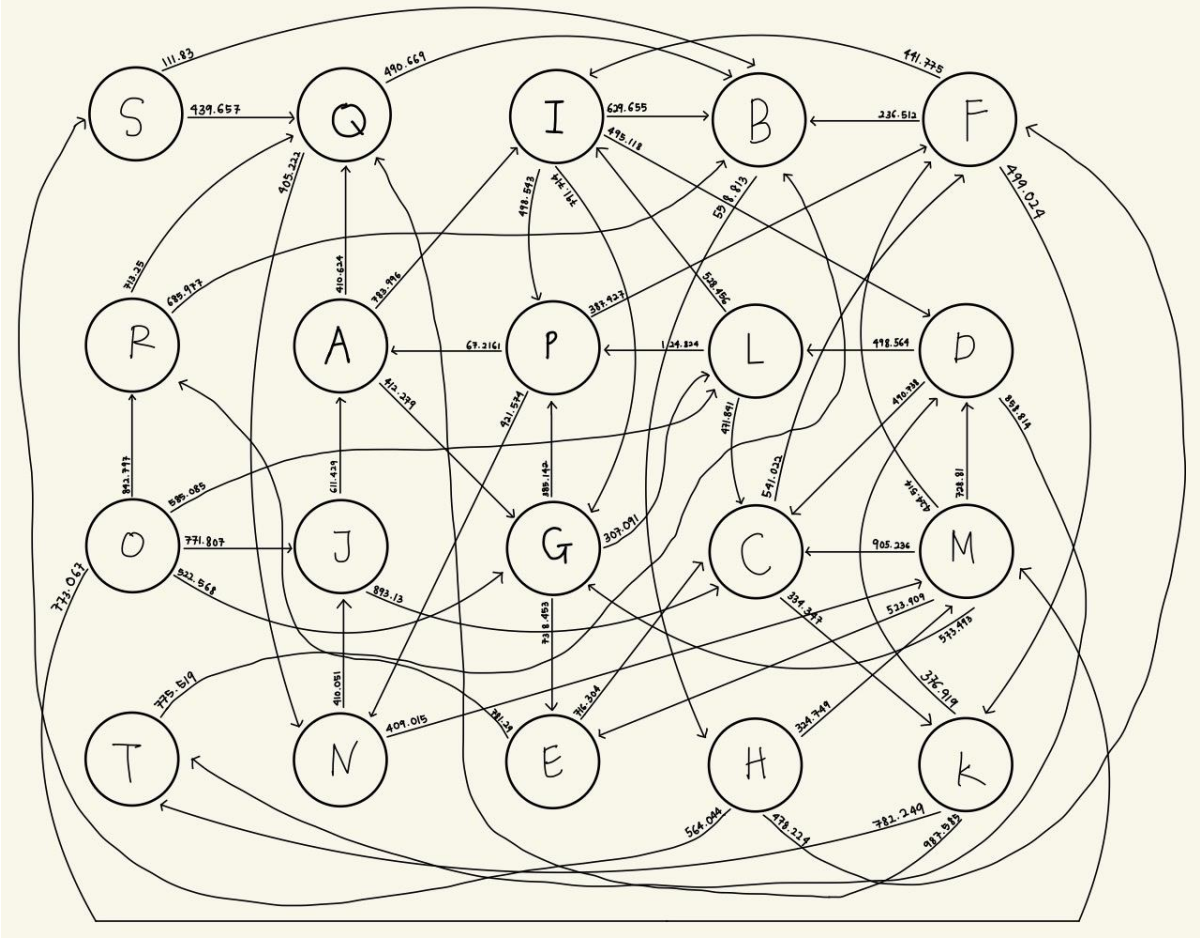
3. Generate Stars:
    - Extracts unique digits from a sum of IDs, seeds the random number generator, and generates stars with random attributes using these digits.

4. Calculate Distance Between Two Stars:
    - This function calculates the Euclidean distance between two stars based on their coordinates.

5. Generate Unique Routes Between Stars:
    - Randomly pairs stars to create routes, ensuring no duplicates.

6. Save Star Dataset to a File:
    - Opens a file for writing, writes star details and route distances, then closes the file.

Expected Graph:



b) Programs

Contain 3 main coding files, main.cpp (for option to choose to generate Dataset 1 or Dataset 2), q1_dataset1.h (logic to generate 6 dataset_n.txt files for Dataset 1) and q1_dataset2.h (logic to generate 1 dataset2_1.txt file for Dataset 2).

i. main.cpp

```
#include <iostream>
#include "q1_dataset1.h"
#include "q1_dataset2.h"
#include <cstdlib>  // For system()
#include <ctime>  // Include the ctime header

void generate_dataset1() {
    // Generate and save datasets
    generate_and_save_datasets();
}

void generate_dataset2() {
    // Get current time as seed
    long long int seed = static_cast<long long int>(time(nullptr));

    // Sum of other group members' ID numbers
    long long int sum_of_ids = 1211101007LL + 1211200107LL + 1221303660LL;
    const int star_count = 20;  // Number of stars to generate

    // Generate stars
    std::vector<Star> stars = generate_stars(star_count, sum_of_ids);

    // Generate routes
    std::vector<std::pair<char, char>> routes = generate_routes(star_count);

    // Save dataset and generate DOT file
    save_star_dataset(stars, routes, "dataset2_1.txt");

    // Convert DOT file to PNG using Graphviz
    int result = system("dot -Tpng stars_graph.dot -o stars_graph.png");
    if (result != 0) {
        std::cerr << "Error: Could not generate PNG file using Graphviz." <<
std::endl;
    } else {
```

```
        std::cout << "PNG file generated successfully: stars_graph.png" <<
std::endl;
    }
}

int main() {
    int choice;
    std::cout << "Select an option:" << std::endl;
    std::cout << "1. Generate dataset 1" << std::endl;
    std::cout << "2. Generate dataset 2" << std::endl;
    std::cout << "Enter your choice: ";
    std::cin >> choice;

    switch (choice) {
        case 1:
            generate_dataset1();
            break;
        case 2:
            generate_dataset2();
            break;
        default:
            std::cerr << "Invalid choice." << std::endl;
            return 1;
    }

    return 0;
}
```

---

ii. q1_dataset1.h (Leader's ID (1211202025) as random seed reference)

```
#ifndef DATASETS_GENERATOR_H
#define DATASETS_GENERATOR_H

#include <iostream>
#include <fstream>
#include <cstdlib>  // For srand() and rand()
#include <string>
#include <vector>
#include <unordered_set>
#include <algorithm>
#include <iterator>
```

```cpp
// Group leader's ID
const long long id_leader = 1211202025;

// Dataset sizes
const int size_dataset[] = {100, 1000, 10000, 100000, 500000, 1000000};

// Function to extract unique digits from a given number
std::vector<int> extract_unique_digits(long long number) {
    std::unordered_set<int> unique_digits;
    while (number > 0) {
        unique_digits.insert(number % 10);
        number /= 10;
    }
    return std::vector<int>(unique_digits.begin(), unique_digits.end());
}

// Function to generate a dataset of random 3-digit numbers
std::vector<int> generate_dataset(int seed, int size, const std::vector<int>&
allowed_digits) {
    std::vector<int> dataset;
    dataset.reserve(size);

    // Seed the random number generator
    srand(seed);

    int num_digits = allowed_digits.size();

    // Generate random numbers
    for (int i = 0; i < size; ++i) {
        int generated_number = 0;
        for (int j = 0; j < 3; ++j) {
            generated_number = generated_number * 10 + allowed_digits[rand() %
num_digits];
        }
        dataset.push_back(generated_number);
    }

    return dataset;
}

// Function to save a dataset to a file
```

```
void save_dataset(const std::vector<int>& dataset, const std::string& filename)
{
    std::ofstream outfile(filename);
    if (outfile.is_open()) {
        std::copy(dataset.begin(), dataset.end(),
std::ostream_iterator<int>(outfile, "\n"));
        outfile.close();
        std::cout << "Dataset with " << dataset.size() << " elements saved to "
<< filename << std::endl;
    } else {
        std::cerr << "Error opening file: " << filename << std::endl;
    }
}

// Function to generate and save multiple datasets
void generate_and_save_datasets() {
    std::vector<int> allowed_digits = extract_unique_digits(id_leader);

    for (int i = 0; i < 6; ++i) {
        std::string filename = "dataset_" + std::to_string(i + 1) + ".txt";
        std::vector<int> dataset = generate_dataset(id_leader + i,
size_dataset[i], allowed_digits);
        save_dataset(dataset, filename);
    }
}

#endif // DATASETS_GENERATOR_H
```

---

iii. q1_dataset2.h (sum of group members' IDs (1211101007 + 1211200107 + 1221303660 = 3643604774) as random seed reference)

```
#ifndef STARS_GENERATOR_H
#define STARS_GENERATOR_H

#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <algorithm>
```

```cpp
#include <unordered_set>

struct Star {
    char name;
    double x, y, z;
    int weight, profit;
};

// Custom random number generator
int custom_rand(const std::vector<int>& allowed_digits) {
    int num_digits = allowed_digits.size();
    return allowed_digits[rand() % num_digits];
}

// Function to extract unique digits from seed
std::vector<int> extract_unique_digits_2(long long seed) {
    std::unordered_set<int> unique_digits;
    while (seed > 0) {
        unique_digits.insert(seed % 10);
        seed /= 10;
    }
    return std::vector<int>(unique_digits.begin(), unique_digits.end());
}

// Function to generate unique random number from allowed digits
int generate_unique_number(const std::vector<int>& allowed_digits, int
num_digits) {
    int generated_number = 0;
    for (int i = 0; i < num_digits; ++i) {
        generated_number = generated_number * 10 + custom_rand(allowed_digits);
    }
    return generated_number;
}

// Function to generate stars
std::vector<Star> generate_stars(int count, long long int sum_of_ids) {
    std::vector<int> allowed_digits = extract_unique_digits_2(sum_of_ids);
    srand(static_cast<unsigned int>(time(0) + sum_of_ids)); // Seed the random
number generator

    std::vector<Star> stars;
    // Generate coordinates and random values for weight and profit for each
star
```

```
    for (char name = 'A'; name < 'A' + count; ++name) {
        Star star;
        star.name = name;
        star.x = generate_unique_number(allowed_digits, 3);  // Random 3-digit
number
        star.y = generate_unique_number(allowed_digits, 3);  // Random 3-digit
number
        star.z = generate_unique_number(allowed_digits, 3);  // Random 3-digit
number
        star.weight = generate_unique_number(allowed_digits, 2);   // Random 2-
digit number
        star.profit = generate_unique_number(allowed_digits, 2);   // Random 2-
digit number
        stars.push_back(star);
    }

    return stars;
}

// Function to calculate distance between two stars
double calculate_distance(double x1, double y1, double z1, double x2, double
y2, double z2) {
    return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2) + pow(z2 - z1, 2));
}

// Function to generate routes ensuring each star connects to at least 3 others
std::vector<std::pair<char, char>> generate_routes(int count) {
    std::vector<std::pair<char, char>> routes;
    int connections = 0;
    while (connections < 54) {  // Ensures 54 unique routes
        char name1 = 'A' + rand() % count;
        char name2 = 'A' + rand() % count;
        if (name1 != name2 && std::find(routes.begin(), routes.end(),
std::make_pair(name1, name2)) == routes.end() &&
            std::find(routes.begin(), routes.end(), std::make_pair(name2,
name1)) == routes.end()) {
            routes.push_back(std::make_pair(name1, name2));
            ++connections;
        }
    }
    return routes;
}
```

```
// Function to save star dataset
void save_star_dataset(const std::vector<Star> &stars, const
std::vector<std::pair<char, char>> &routes, const std::string &filename) {
    std::ofstream outfile(filename);
    if (!outfile.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    // Save star details
    for (const auto &star : stars) {
        outfile << "Star " << star.name << " " << star.x << " " << star.y << "
" << star.z << " " << star.weight << " " << star.profit << std::endl;
    }

    // Save route details with distances
    for (const auto &route : routes) {
        Star star1, star2;
        for (const auto &star : stars) {
            if (star.name == route.first) star1 = star;
            if (star.name == route.second) star2 = star;
        }
        double distance = calculate_distance(star1.x, star1.y, star1.z,
star2.x, star2.y, star2.z);
        outfile << "Route " << route.first << "-" << route.second << "
Distance: " << distance << std::endl;
    }

    outfile.close();
    std::cout << "Star dataset saved to " << filename << std::endl;
}

#endif // STARS_GENERATOR_H
```

2. Heap Sort and Selection Sort

a) Algorithms

i. Heap Sort

Problem Statement:

Store the data in a priority queue using a heap. Store sorted dataset(s)/output(s) in e.g. txt file(s). Record the time to insert all data into the priority queue. Dequeue the data to test the timing of the priority queue. Plot the graph of timing vs dataset size and discuss about the time and space complexity of the algorithm.

Algorithm Functions and Description:

- readDataset: Reads a number of integers from a text file and stores them in a vector
- writeSortedDataset: Writes the contents of a sorted vector to a text file
- heapify: Ensures that a subtree with root at a given index *i* satisfies the max-heap property, meaning that the value of the root is greater than or equal to the values of its children.
- heapSort: Sort an array (or vector) of integers in ascending order

Algorithm Execution:

1. **Initialization**:
    - The main function initializes vectors for filenames (filenames) and dataset sizes (datasetSizes) to iterate through different datasets.
    - It uses a loop to process each dataset sequentially.
2. **Reading and Preparing Datasets:**
    - Inside the loop, the code reads a dataset from a file (filenames[i]) using readDataset, which populates a vector (dataset) with integers from the file.
    - Each dataset is read according to its specified size (datasetSizes[i]).
3. **Heap Sort Execution**:

    Heapify Function: Defined to maintain the max heap property of an array.
    - It identifies the largest element among a root and its two children.
    - If necessary, it swaps elements to ensure the largest element is at the root.

    Heap Sort Function:

    Build Max Heap:
    - Constructs a max heap from the array by calling heapify on each non-leaf node, starting from the last non-leaf node down to the root.

    Heap Sort Proper:
    - Continuously extracts the largest element from the heap (root) and places it at the end of the array.

- Rebuilds the heap structure for the remaining elements using heapify.
- Repeats until the entire array is sorted.

4. **Timing and Output**

   Timing Heap Sort:

   - Records the start time (startHeap) before invoking heapSort.
   - Measures the duration of heapSort using chrono::high_resolution_clock.
   - Records the end time (endHeap) after heapSort.

   Writing Results:

   - Saves the sorted dataset (heapSortedDataset) to a text file (heapSortOutputFilename) using writeSortedDataset.
   - Prints the timing information for heap sort execution (heapDuration) to standard output.

ii. Selection Sort

Problem Statement:

Sort the dataset(s) using selection sort. Store sorted dataset(s)/output(s) in e.g. txt file(s). Record the time to insert all data into the priority queue. Dequeue the data to test the timing of the priority queue. Plot the graph of timing vs dataset size and discuss about the time and space complexity of the algorithm.

Algorithm Functions and Description:

- readDataset: Reads a number of integers from a text file and stores them in a vector
- writeSortedDataset: Writes the contents of a sorted vector to a text file
- selectionSort: Sorts an array (or vector) of integers in ascending order using the selection sort algorithm.

Algorithm Execution:

1. **Initialization:**
    - The main function initializes vectors for filenames (filenames) and dataset sizes (datasetSizes) to iterate through different datasets.
    - It uses a loop to process each dataset sequentially.Inside the loop, the code reads a dataset from a file (filenames[i]) using readDataset, which populates a vector (dataset) with integers from the file.
    - Each dataset is read according to its specified size (datasetSizes[i]).
2. **Reading and Preparing Datasets:**
    - Inside the loop, the code reads a dataset from a file (filenames[i]) using readDataset, which populates a vector (dataset) with integers from the file.
    - Each dataset is read according to its specified size (datasetSizes[i]).
3. **Selection Sort Function**
    - Finds the minimum element from the unsorted part of the array in each iteration.
    - Swaps it with the element at the beginning of the unsorted part.
    - Continues until the entire array is sorted.
4. **Timing and Output**

    Timing Selection Sort:
    - Records the start time (startSelection) before invoking selectionSort.

- Measures the duration of selectionSort using chrono::high_resolution_clock.
- Records the end time (endSelection) after selectionSort

Writing Results:

- Saves the sorted dataset (`selectionSortedDataset`) to a text file (`selectionSortOutputFilename`) using `writeSortedDataset`.
- Prints the timing information for selection sort execution (`selectionDuration`) to standard output.

a) Programs

i) sorting.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <chrono>

using namespace std;

// Function to read dataset from a text file
vector<int> readDataset(const string& filename, int size) {
    vector<int> dataset;
    dataset.reserve(size);
    ifstream inputFile(filename);
    int value;
    for (int i = 0; i < size && inputFile >> value; ++i) {
        dataset.push_back(value);
    }
    inputFile.close();
    return dataset;
}

// Function to write sorted dataset to a text file
void writeSortedDataset(const vector<int>& sortedDataset, const string&
outputFilename) {
    ofstream outputFile(outputFilename);
    for (int value : sortedDataset) {
        outputFile << value << "\n";
    }
    outputFile.close();
}
```

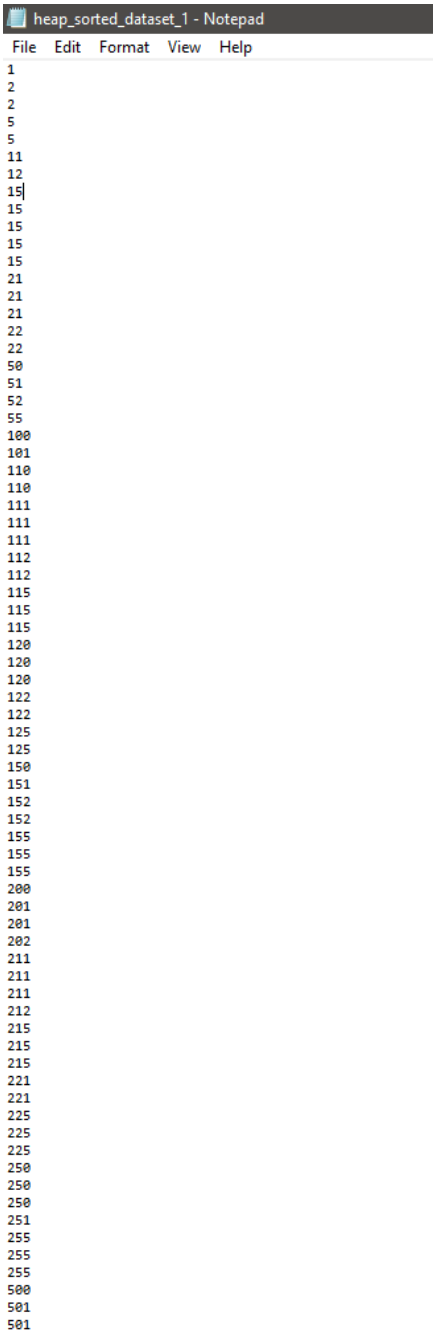
```cpp
// Heapify function for heap sort
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// Heap sort function
void heapSort(vector<int>& arr) {
    int n = arr.size();

    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Heap sort
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// Selection sort function
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
```

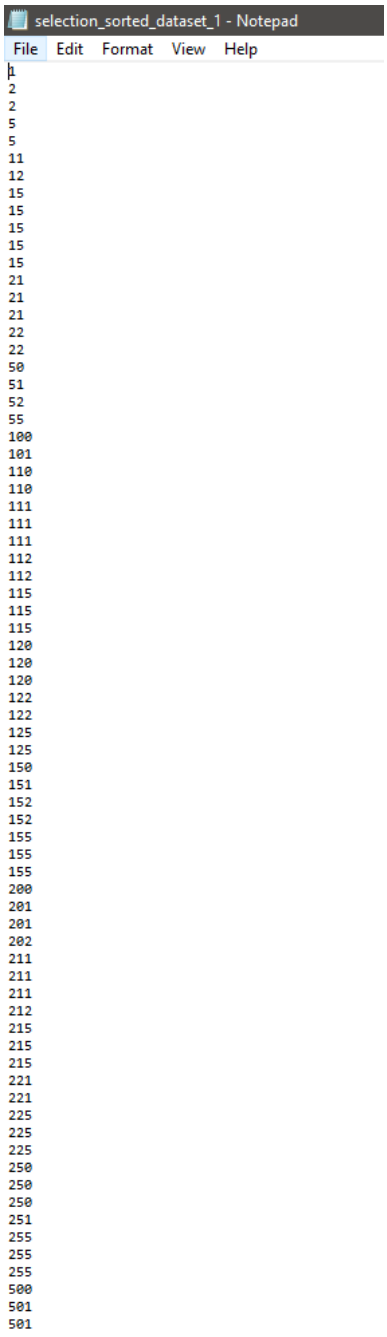
```
        swap(arr[i], arr[minIndex]);
    }
}

int main() {
    vector<string> filenames = {"dataset_1.txt", "dataset_2.txt",
"dataset_3.txt", "dataset_4.txt", "dataset_5.txt", "dataset_6.txt"};
    vector<int> datasetSizes = {100, 1000, 10000, 100000, 500000, 1000000};

    for (int i = 0; i < 6; ++i) {
        // Read dataset from text file
        vector<int> dataset = readDataset(filenames[i], datasetSizes[i]);

        // Perform heap sort
        vector<int> heapSortedDataset = dataset;
        auto startHeap = chrono::high_resolution_clock::now();
        heapSort(heapSortedDataset);
        auto endHeap = chrono::high_resolution_clock::now();
        chrono::duration<double> heapDuration = endHeap - startHeap;

        // Write heap sorted dataset to text file
        string heapSortOutputFilename = "heap_sorted_dataset_" + to_string(i + 1)
+ ".txt";
        writeSortedDataset(heapSortedDataset, heapSortOutputFilename);

        // Perform selection sort
        vector<int> selectionSortedDataset = dataset;
        auto startSelection = chrono::high_resolution_clock::now();
        selectionSort(selectionSortedDataset);
        auto endSelection = chrono::high_resolution_clock::now();
        chrono::duration<double> selectionDuration = endSelection -
startSelection;

        // Write selection sorted dataset to text file
        string selectionSortOutputFilename = "selection_sorted_dataset_" +
to_string(i + 1) + ".txt";
        writeSortedDataset(selectionSortedDataset, selectionSortOutputFilename);

        // Print timing information
        cout << "Dataset " << (i + 1) << ":\n";
        cout << "Heap Sort Time: " << heapDuration.count() << " seconds\n";
        cout << "Selection Sort Time: " << selectionDuration.count() << "
seconds\n\n";
```

```
    }

    return 0;
}
```

b) Experimental results

heap_sorted_dataset_1 and selection_sorted_dataset1:

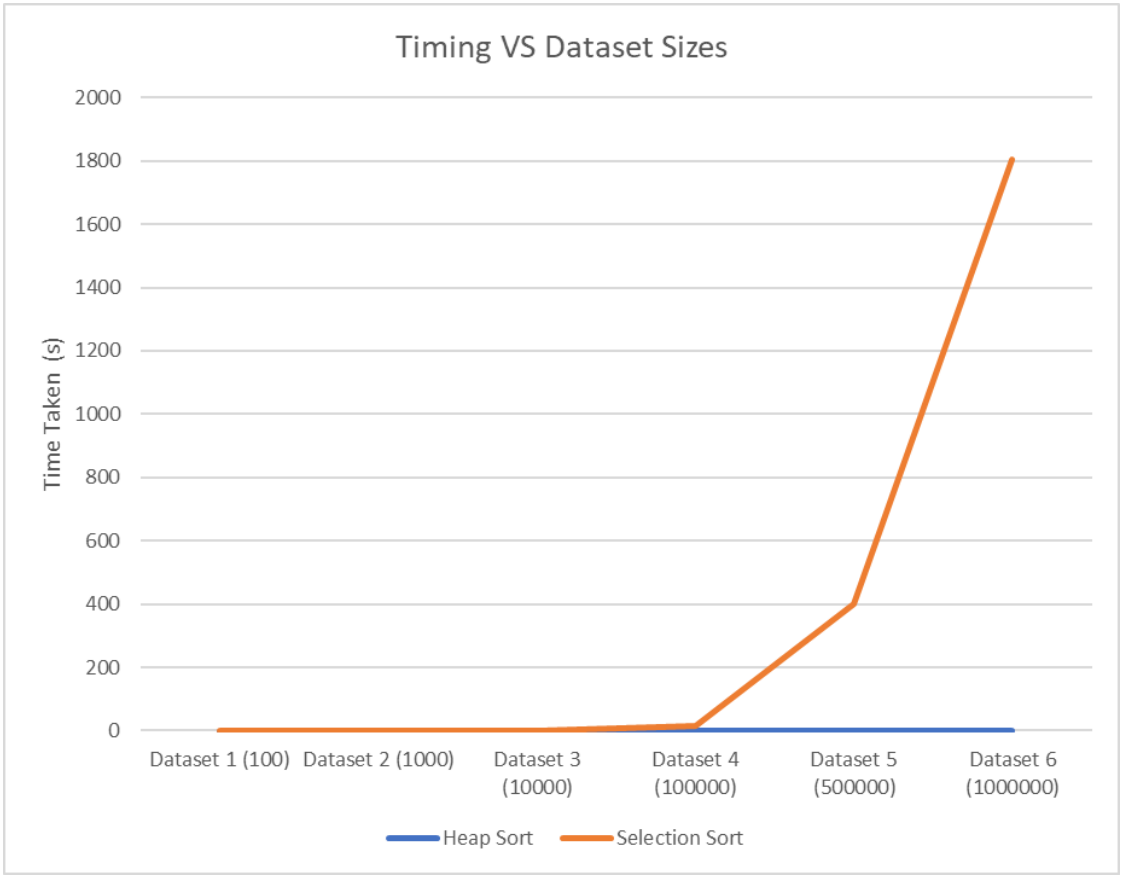




The time taken to complete sorting of each dataset for Heap Sort and Selection Sort is shown below:

```
Dataset 1:
Heap Sort Time: 0 seconds
Selection Sort Time: 0 seconds

Dataset 2:
Heap Sort Time: 0 seconds
Selection Sort Time: 0.002001 seconds

Dataset 3:
Heap Sort Time: 0.002001 seconds
Selection Sort Time: 0.161037 seconds

Dataset 4:
Heap Sort Time: 0.029006 seconds
Selection Sort Time: 16.3985 seconds

Dataset 5:
Heap Sort Time: 0.164037 seconds
Selection Sort Time: 398.518 seconds

Dataset 6:
Heap Sort Time: 0.348079 seconds
Selection Sort Time: 1802.42 seconds
```

Plotted graph of the time taken over the size of datasets is as shown below:



c) Discussion

Time Complexity:

- Heap Sort:
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
- Selection Sort:
  - Best Case: $O(n^2)$
  - Average Case: $O(n^2)$
  - Worst Case: $O(n^2)$

Space Complexity:

- Heap Sort: O(1)
- Selection Sort: O(1)

Conclusion: All of the above proves that Heap Sort is far better than Selection Sort, since it has a better Time Complexity and also is proven with a plotted graph that it takes less time to generate the sorted dataset.

3. Shortest Paths and Minimum Spanning Tree

a) Algorithms

i. Dijkstra- Shortest Paths

Problem Statement:

Write a program to identify the shortest paths from Star A to the other stars using Dijkstra's Algorithm. Then display the shortest distance to each star (store the output(s) in e.g. txt file(s)). Draw the graph representing the shortest paths. Discuss about the time and space complexity of the algorithm.

Algorithm Description:

Data Structures

- **DijkstraStar Struct**: Represents a star with properties such as name, coordinates (x, y, z), weight, and profit. This structure is used to store information about each vertex in the graph.
- **Edge Struct**: Represents an edge in the graph, consisting of a destination star (to) and the distance associated with that edge.
- **PathInfo Struct**: Stores information about the shortest path to a particular star, including the shortest distance and the sequence of stars (path) leading to that star.
- **std::unordered_map<char, std::vector<Edge>> graph**: Represents the graph itself using an unordered map where each key (char) represents a star and its corresponding value is a vector of Edge objects representing edges to adjacent stars.

Functions

- **read_star_dataset**: Reads a dataset from a file (filename) containing information about stars and routes (edges). Stars are read into a vector of DijkstraStar structs, and routes are used to populate the graph representation.
- **dijkstra_calculate_distance**: Computes the Euclidean distance between two stars based on their coordinates (x, y, z).
- **dijkstra**: The core function that implements Dijkstra's algorithm. It takes the graph and a starting star (start) as inputs and computes the shortest paths from start to all other stars using a priority queue (pq). It updates the shortest distances (distances) and paths as it explores each vertex.
- **save_distances**: Saves the computed shortest distances and paths to a file (filename).

Algorithm Execution

Initialization: Initialize a priority queue (pq) to keep track of stars to be processed based on their tentative distances from the start star. Initialize distances with initial values (infinity for all except the start star which is star A itself initialized to 0).

Processing Loop: Continue processing stars from pq until all reachable stars have been processed.

- Extract the star with the smallest tentative distance from pq.
- Update the tentative distances of its neighboring stars if a shorter path is found.
- Update the priority queue with the new distances and paths.

Completion: Once pq is empty, distances contains the shortest paths and distances from the start star to all other stars.

ii. Kruskal- Minimum Spanning Tree

Problem Statement:

Write a program to identify the Minimum Spanning Tree using Kruskal's Algorithm. Display the edges of the tree and draw the graph representing the Minimum Spanning Tree (store the output(s) in e.g. txt file(s)). Discuss about the time and space complexity of the algorithm.

Algorithm Description:

Data Structures:

- **KruskalEdge Struct**: Explain the structure used to represent edges between stars, consisting of from, to, and distance.
- **Union-Find Data Structure**: Detail the UnionFind struct used for efficient cycle detection and union operations, crucial for implementing Kruskal's algorithm.

Functions:

- **read_kruskal_dataset**: Describe how this function reads dataset information from a file (dataset2_1.txt) and parses it into KruskalEdge structures.
- **kruskal**: Explain the implementation of Kruskal's algorithm using the sorted edges and the Union-Find data structure to construct the MST.
- **save_mst**: Outline how the MST computed by Kruskal's algorithm is saved to a file (minimum_spanning_tree.txt).

Algorithm Execution:

Initialization:

- The algorithm begins by initializing an empty vector mst to hold edges that will form the Minimum Spanning Tree.
- The dataset is read from dataset2_1.txt to obtain a list of edges (from, to, distance) using the read_kruskal_dataset function.

Union-Find Data Structure Initialization:

- A UnionFind structure is initialized with vertex_count, where each vertex starts as its own set. This structure efficiently manages union and find operations, crucial for cycle detection during MST construction.

Main Algorithm Loop:

- The algorithm iterates through each edge in the sorted_edges vector:
    - edge is converted from star names (from, to) to indices (u, v) using the get_index lambda function, simplifying graph operations.
    - uf.find(u) and uf.find(v) determine if vertices u and v belong to the same set. If they don't (uf.find(u) != uf.find(v)), adding edge won't form a cycle.
    - If no cycle is detected:
        - uf.union_sets(u, v) merges the sets containing u and v, ensuring efficient union operation using union by rank and path compression techniques.
        - The edge edge is added to mst, progressing towards constructing the MST.
        - The loop exits early if mst contains vertex_count - 1 edges, completing the MST.

Output:

- The computed MST stored in mst is saved to minimum_spanning_tree.txt using the save_mst function. This file contains the minimal set of edges connecting all vertices without cycles.

b) Programs

i. Dijkstra – Shortest Path

dijkstra_operations.h

```
#ifndef DIJKSTRA_OPERATIONS_H
#define DIJKSTRA_OPERATIONS_H

#include <iostream>
```

```
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <queue>
#include <unordered_map>
#include <limits>
#include <cmath>
#include <algorithm>

// Define DijkstraStar struct
struct DijkstraStar {
   char name;
   double x, y, z; // x-axis, y-axis, z-axis
   int weight, profit;
};

// Define Edge struct for graph representation
struct Edge {
   char to; // star destination
   double distance;
};

// Define PathInfo struct to store distance and path
struct PathInfo {
   double distance;
   std::vector<char> path;
};

// Function to calculate distance between two DijkstraStars
double dijkstra_calculate_distance(double x1, double y1, double z1, double x2,
double y2, double z2) {
   return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2) + pow(z2 - z1, 2));
}

// Function to read star dataset for Dijkstra's algorithm
void read_star_dataset(const std::string &filename, std::vector<DijkstraStar>
&stars, std::unordered_map<char, std::vector<Edge>> &graph) {
   std::ifstream infile(filename);
   if (!infile.is_open()) {
      std::cerr << "Error opening file: " << filename << std::endl;
      return;
   }
```

```
    std::string line;
    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        std::string type;
        iss >> type;

        if (type == "Star") {
            DijkstraStar star;
            iss >> star.name >> star.x >> star.y >> star.z >> star.weight >>
star.profit;
            stars.push_back(star);
        } else if (type == "Route") {
            char from, to;
            std::string delimiter;
            double distance;
            iss >> from >> delimiter >> to >> delimiter >> distance;
            graph[from].push_back({to, distance});
            graph[to].push_back({from, distance});
        }
    }

    infile.close();
}

// Function to apply Dijkstra's Algorithm
std::unordered_map<char, PathInfo> dijkstra(const std::unordered_map<char,
std::vector<Edge>> &graph, char start) {
    // Store shortest distance and path from start star to other stars
    std::unordered_map<char, PathInfo> distances;
    for (const auto &node : graph) {
        // Stars unreachable from start
        distances[node.first] = {std::numeric_limits<double>::infinity(), {}};
    }
    // Set distance from start to itself as 0
    distances[start] = {0, {start}};

    // Compare star (char) and distances (double)
    auto compare = [](const std::pair<char, double> &a, const std::pair<char,
double> &b) {
        return a.second > b.second;
    };
    std::priority_queue<std::pair<char, double>, std::vector<std::pair<char,
double>>, decltype(compare)> pq(compare);
    pq.push({start, 0});
```

```
    // Iterate until pq is empty
    while (!pq.empty()) {
        // Retrieve star with smallest distance from pq
        char current = pq.top().first;
        double current_distance = pq.top().second;
        pq.pop();

        // Check if current star is greater than currently known shortest distance
        if (current_distance > distances[current].distance) {
            continue;
        }

        for (const auto &edge : graph.at(current)) {
            double new_distance = current_distance + edge.distance;
            if (new_distance < distances[edge.to].distance) {
                distances[edge.to].distance = new_distance;
                distances[edge.to].path = distances[current].path;
                distances[edge.to].path.push_back(edge.to);
                pq.push({edge.to, new_distance});
            }
        }
    }

    return distances;
}

// Function to save distances and paths to a file
void save_distances(const std::unordered_map<char, PathInfo> &distances,
const std::string &filename) {
    std::ofstream outfile(filename);
    if (!outfile.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    for (const auto &entry : distances) {
        outfile << "Shortest distance from Star A to Star " << entry.first << " is " <<
entry.second.distance << ", path: {";
        for (const auto &star : entry.second.path) {
            outfile << star << (star == entry.second.path.back() ? "" : ", ");
        }
        outfile << "}" << std::endl;
    }
```

```
    outfile.close();
}

#endif // DIJKSTRA_OPERATIONS_H
```

shortest_paths.cpp

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <chrono>
#include "dijkstra_operations.h"

void save_shortest_paths(const std::unordered_map<char, PathInfo>&
distances, const std::vector<DijkstraStar>& stars, const std::string& filename) {
    std::ofstream outfile(filename);
    if (!outfile.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    // Ensure all stars are included in the output
    for (const auto& star : stars) {
        char star_name = star.name;
        if (distances.find(star_name) != distances.end()) {
            const auto& info = distances.at(star_name);
            outfile << "Shortest distance from Star A to Star " << star_name << " is "
<< info.distance <<std::endl;

        } else {
            outfile << "Star " << star_name << " is unreachable from Star A." <<
std::endl;
        }
    }

    outfile.close();
}


int main() {
    std::vector<DijkstraStar> stars;
```

```
    std::unordered_map<char, std::vector<Edge>> graph;

    // Start timing for the entire program
    auto start_program = std::chrono::steady_clock::now();

    // Read dataset
    read_star_dataset("dataset2_1.txt", stars, graph);

    // Find shortest paths from Star A
    std::unordered_map<char, PathInfo> distances = dijkstra(graph, 'A');

    // Save shortest paths to file
    save_shortest_paths(distances, stars, "shortest_paths.txt");

    // End timing for the entire program
    auto end_program = std::chrono::steady_clock::now();

    // Calculate total duration
    auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end_program -
start_program);
    std::cout << "Result is saved to shortest_paths.txt" <<std::endl;

    // Print execution time of the whole program
    std::cout << "Total execution time: " << duration.count() << " milliseconds."
<< std::endl;

    return 0;
}
```

ii. Kruskal – Minimum Spanning Tree

Kruskal.h

```
#ifndef KRUSKAL_MINIMUM_SPANNING_TREE_H
#define KRUSKAL_MINIMUM_SPANNING_TREE_H

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
```

```
#include <unordered_map>
#include "q1_dataset2.h"

struct KruskalEdge {
   char from;
   char to;
   double distance;
};

// Union-Find data structure
struct UnionFind {
   std::vector<int> parent, rank;

   UnionFind(int n) {
      parent.resize(n);
      rank.resize(n, 0);
      for (int i = 0; i < n; ++i) {
         parent[i] = i;
      }
   }

   int find(int u) {
      if (parent[u] != u) {
         parent[u] = find(parent[u]); // Path compression
      }
      return parent[u];
   }

   void union_sets(int u, int v) {
      int root_u = find(u);
      int root_v = find(v);
      if (root_u != root_v) {
         if (rank[root_u] > rank[root_v]) {
            parent[root_v] = root_u;
         } else if (rank[root_u] < rank[root_v]) {
            parent[root_u] = root_v;
         } else {
            parent[root_v] = root_u;
            rank[root_u]++;
         }
      }
   }
};
```

```
// Function to read dataset for Kruskal's algorithm
std::vector<KruskalEdge> read_kruskal_dataset(const std::string &filename) {
   std::ifstream infile(filename);
   std::vector<KruskalEdge> edges;

   if (!infile.is_open()) {
      std::cerr << "Error opening file: " << filename << std::endl;
      return edges; // Return an empty vector if file cannot be opened
   }

   std::string line;
   while (std::getline(infile, line)) {
      char from, to;
      double distance;
      if (sscanf(line.c_str(), "Route %c-%c Distance: %lf", &from, &to, &distance)
== 3) {
         edges.push_back({from, to, distance});
      }
   }

   infile.close();
   return edges;
}

// Define Kruskal's algorithm implementation
std::vector<KruskalEdge> kruskal(const std::vector<KruskalEdge> &edges, int
vertex_count) {
   std::vector<KruskalEdge> mst;

   // Sort the edges by distance
   std::vector<KruskalEdge> sorted_edges = edges;
   std::sort(sorted_edges.begin(), sorted_edges.end(), [](const KruskalEdge &a,
const KruskalEdge &b) {
      return a.distance < b.distance;
   });

   UnionFind uf(vertex_count);

   auto get_index = [](char name) { return name - 'A'; }; // Convert star name to
index

   for (const auto &edge : sorted_edges) {
      int u = get_index(edge.from);
      int v = get_index(edge.to);
```

```
        if (uf.find(u) != uf.find(v)) {
            uf.union_sets(u, v);
            mst.push_back(edge);
            if (mst.size() == vertex_count - 1) break; // Stop when MST has n-1
edges
        }
    }

    return mst;
}

void save_mst(const std::vector<KruskalEdge> &mst, const std::string
&filename) {
    std::ofstream outfile(filename);
    if (!outfile.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    for (const auto &edge : mst) {
        outfile << "Connected Stars: " << edge.from << " - " << edge.to << "
Distance: " << edge.distance << std::endl;
    }

    outfile.close();
    std::cout << "MST saved to " << filename << std::endl;
}

#endif // KRUSKAL_MINIMUM_SPANNING_TREE_H
```

mst.cpp

```
#include <iostream>
#include <chrono> // For timing

#include "q1_dataset2.h"
#include "kruskal.h"

void generate_dataset2()
{
    // Sum of other group members' ID numbers
    long long int sum_of_ids = 1211101007LL + 1211200107LL + 1221303660LL;
    const int star_count = 20; // Number of stars to generate
```

```
    // Generate stars
    std::vector<Star> stars = generate_stars(star_count, sum_of_ids);

    // Generate routes
    std::vector<std::pair<char, char>> routes = generate_routes(star_count);

    // Save dataset
    save_star_dataset(stars, routes, "dataset2_1.txt");
}

void find_minimum_spanning_tree()
{
    // Read dataset
    std::vector<KruskalEdge> edges = read_kruskal_dataset("dataset2_1.txt");

    // Find MST using Kruskal's algorithm
    auto start_time = std::chrono::high_resolution_clock::now(); // Start timing
    std::vector<KruskalEdge> mst = kruskal(edges, 20);          // Assuming there are 20
stars
    auto end_time = std::chrono::high_resolution_clock::now();   // End timing

    // Calculate duration
    std::chrono::duration<double> duration = end_time - start_time;

    // Save MST
    save_mst(mst, "minimum_spanning_tree.txt");

    // Output execution time
    std::cout << "Execution time: " << duration.count() << " seconds" << std::endl;
}

int main()
{

    find_minimum_spanning_tree(); // Call the function to find minimum spanning tree

    return 0;
}
```

c) Experimental results

   i. Dijkstra – Shortest Paths

Dijkstra > shortest_paths.txt

```
1   Shortest distance from Star A to Star A is 0
2   Shortest distance from Star A to Star B is 969.437
3   Shortest distance from Star A to Star C is 744.971
4   Shortest distance from Star A to Star D is 410.624
5   Shortest distance from Star A to Star E is 1126.93
6   Shortest distance from Star A to Star F is 647.136
7   Shortest distance from Star A to Star G is 477.84
8   Shortest distance from Star A to Star H is 735.373
9   Shortest distance from Star A to Star I is 905.742
10  Shortest distance from Star A to Star J is 1022.05
11  Shortest distance from Star A to Star K is 787.543
12  Shortest distance from Star A to Star L is 535.448
13  Shortest distance from Star A to Star M is 835.138
14  Shortest distance from Star A to Star N is 819.639
15  Shortest distance from Star A to Star O is 933.192
16  Shortest distance from Star A to Star P is 798.051
17  Shortest distance from Star A to Star Q is 815.846
18  Shortest distance from Star A to Star R is 1096.6
19  Shortest distance from Star A to Star S is 522.454
20  Shortest distance from Star A to Star T is 1186.14
21
```

Figure 3.i.i shows the output for shortest paths

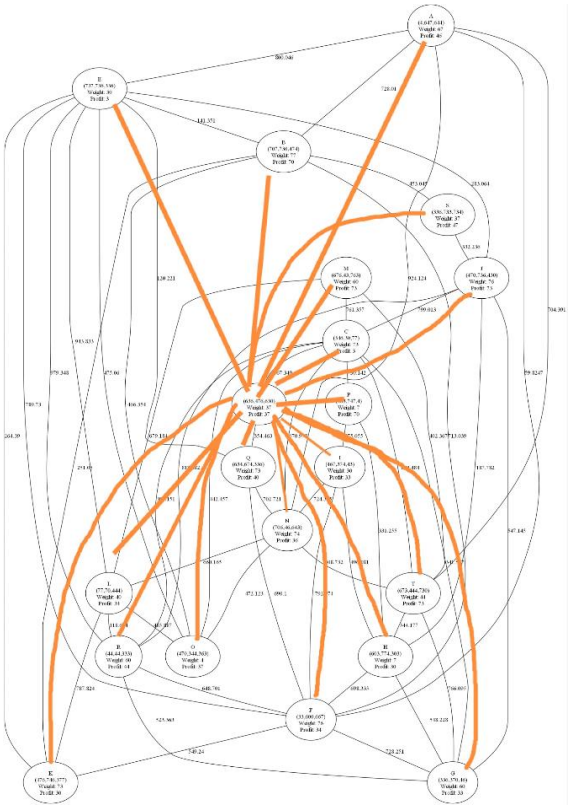


Figure 3.i.ii shows the representation for shortest path

ii. Kruskal – Minimum Spanning Tree

Kruskal > minimum_spanning_tree.txt

```
1   Connected Stars: G - A Distance: 67.2161
2   Connected Stars: S - B Distance: 111.83
3   Connected Stars: L - P Distance: 124.824
4   Connected Stars: F - B Distance: 236.512
5   Connected Stars: G - L Distance: 307.091
6   Connected Stars: H - M Distance: 324.749
7   Connected Stars: C - K Distance: 334.347
8   Connected Stars: K - D Distance: 376.919
9   Connected Stars: P - F Distance: 387.427
10  Connected Stars: Q - N Distance: 405.222
11  Connected Stars: N - M Distance: 409.015
12  Connected Stars: N - J Distance: 410.051
13  Connected Stars: A - Q Distance: 410.624
14  Connected Stars: F - I Distance: 441.775
15  Connected Stars: L - C Distance: 471.841
16  Connected Stars: O - G Distance: 522.568
17  Connected Stars: M - E Distance: 523.909
18  Connected Stars: R - B Distance: 685.977
19  Connected Stars: T - B Distance: 775.519
20
```

Figure 3.ii.i shows the output for minimum spanning tree

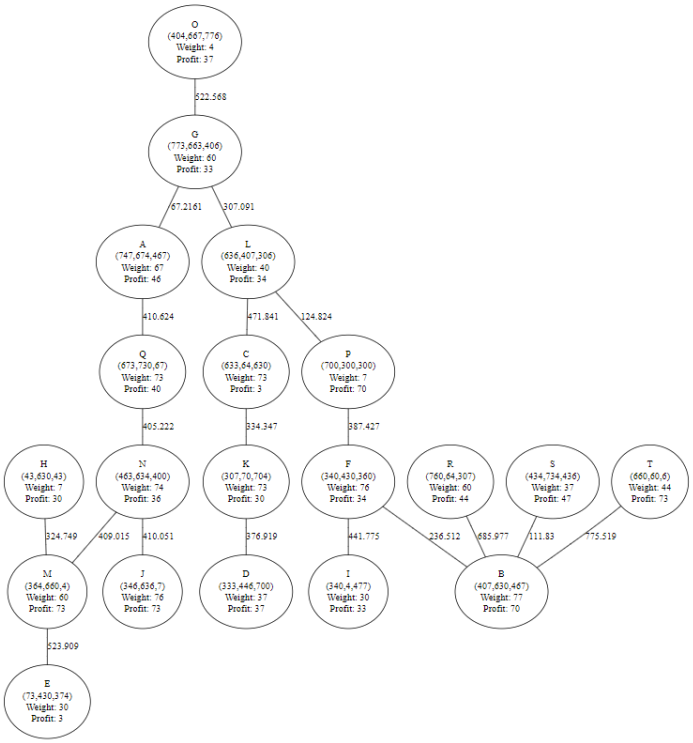


Figure 3.ii.ii shows the representation of minimum spanning tree

d) Discussion

i. Dijkstra – Shortest Path

Time Complexity

1. **Reading Dataset (read_star_dataset function):**
    - Time complexity: O(S+R), where S is the number of stars and R is the number of routes.
    - This is because each star and route entry is processed once.
2. **Dijkstra's Algorithm (dijkstra function):**
    - Time complexity: O((S+R) log S) using a priority queue (min-heap), where S is the number of stars and R is the number of routes.
    - Each star can be pushed and popped from the priority queue at most once, and updating distances and paths involves operations that depend logarithmically on the number of stars.
3. **Overall Time Complexity (Main Functionality):**
    - The main operations contributing to time complexity are reading the dataset O(S+R), running Dijkstra's algorithm O((S+R) log S) and saving results O(S)
    - Therefore, the overall time complexity is dominated by O((S+R) log S)

Space Complexity

1. **Data Structures (stars and graph):**
    - Space complexity: O(S+R), where S is the number of stars and R is the number of routes
2. **Dijkstra's Algorithm (dijkstra function):**
    - Space complexity: O(S) for storing distances and paths, and O(S) for the priority queue.
    - The priority queue can hold up to S stars, and the distances and paths are stored for each star.
3. **Overall Space Complexity:**
    - The overall space complexity is dominated by the data structures used for stars and routes O(S+R), along with O(S) space required by Dijkstra's algorithm.

Conclusion

- **Time Complexity:** O((S+R) log S)
- **Space Complexity:** O(S+R)

ii. Kruskal – Minimum Spanning Tree

Time Complexity

1. **Sorting Edges:**
    - The edges are sorted based on their distance. Sorting is done using std::sort, which typically has a time complexity of O(E log E), where E is the number of edges.
2. **Union-Find Operations:**
    - Kruskal's algorithm utilizes the Union-Find data structure to efficiently manage and merge sets of vertices. The time complexity of the find and union_sets operations can be O(α(V)), where V is the number of vertices, and α is the inverse Ackermann function, which grows very slowly and is practically constant for all practical purposes.
3. **Overall Complexity:**
    - Sorting the edges dominates the time complexity: O(E log E).
    - Union-Find operations in Kruskal's algorithm add O(Eα(V)).
    - Hence, the overall time complexity is O(E log E + Eα(V)).

Space Complexity Analysis

1. **Union-Find Data Structure:**
    - Space complexity for Union-Find is O(V) for storing parent and rank arrays.
2. **Additional Space:**
    - Additional space is used for storing the vector of edges, which contributes O(E) space complexity.
3. **Overall Space Complexity:**
    - The space complexity is O(V + E).

Summary

- **Time Complexity:** O(E log E + Eα(V))
- **Space Complexity:** O(V + E)

4. Dynamic Programming

a) Algorithms

Problem Statement

Assume that each participant travels in a spaceship which have maximum capacity of conquering 800kg of stars. Write a program to identify the set of star(s) to visit without returning to beginning star location using the 0/1 Knapsack Algorithm (assume that once the participant reach a star, participant has to conquer and plunder the full weight of that star). Draft out the resulting matrix and the list of stars to visit, with the weights and profits from each star to be winner of the star conquer quest (store the output(s) in e.g. txt file(s)). Discuss about the time and space complexity of the algorithm

Algorithm Description

The solution utilizes Dynamic Programming (DP) to efficiently solve the 0/1 Knapsack problem:

1. **Initialization**:
    - Stars are read from a file (*dataset2_1.txt*), and each star's characteristics (name, coordinates, weight, profit) are stored in a vector.
    - An unordered set (*star_names*) ensures that stars with duplicate names are not added redundantly.

2. **DP Table Setup**:
    - A DP table (dp) is initialized with dimensions (*n + 1*) x (*capacity + 1*), where n is the number of stars.
    - dp[i][w] represents the maximum profit achievable with the first i stars using a maximum weight capacity of w.

3. **DP Table Population**:
    - Iterate through each star and for each possible capacity *w*.
    - If including the current star does not exceed capacity (*stars[i - 1].weight <= w*), decide whether to take the maximum of either including the current star or excluding it.
    - Update *dp[i][w]* accordingly.

4. **Backtracking**:

- Once the DP table is populated, trace back from *dp[n][capacity]* to determine which stars were selected.
- Collect these stars into *selected_stars* while adjusting the remaining capacity *w*.

5. **Output**:
   - Calculate the total profit obtained from the selected stars.
   - Save the results to *knapsack_result.txt*, including the total weight and profit, the list of selected stars with their respective weights and profits, and the DP table for visualization.

Complexity Analysis

- **Time Complexity**: *O(n*W)*, where n is the number of stars and W is the capacity. This is because the algorithm iterates through each star and each possible weight capacity up to W.
- **Space Complexity**: *O(n*W)*, primarily due to the storage space required for the DP table (dp). Here, sizeof(int) bytes are allocated for each entry in the table.

b) Programs

knapsack.cpp

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <tuple>
#include <iomanip>
#include <algorithm>
#include <unordered_set>
#include <chrono>

// Structure to represent a Star
struct Star {
    char name;
    double x, y, z;
    int weight, profit;
};
```

```cpp
// Function to read stars from a file
std::vector<Star> read_stars(const std::string &filename) {
    std::ifstream infile(filename);
    std::vector<Star> stars;
    std::unordered_set<char> star_names; // To keep track of added star names

    if (infile.is_open()) {
        std::string line;
        while (std::getline(infile, line)) {
            Star star;
            std::sscanf(line.c_str(), "Star %c %lf %lf %lf %d %d", &star.name,
&star.x, &star.y, &star.z, &star.weight, &star.profit);

            // Check if the star name has already been added
            if (star_names.find(star.name) == star_names.end()) {
                stars.push_back(star);
                star_names.insert(star.name); // Add the star name to the set
            }
        }
        infile.close();
    } else {
        std::cerr << "Error opening file: " << filename << std::endl;
    }
    return stars;
}

// Function to solve the 0/1 Knapsack problem using Dynamic Programming
std::tuple<int,        std::vector<Star>,        std::vector<std::vector<int>>>
knapsack(const std::vector<Star> &stars, int capacity) {
    int n = stars.size();
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));

    // Build the DP table
    for (int i = 1; i <= n; ++i) {
        for (int w = 1; w <= capacity; ++w) {
            if (stars[i - 1].weight <= w) {
                dp[i][w] = std::max(dp[i - 1][w], dp[i - 1][w - stars[i -
1].weight] + stars[i - 1].profit);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
```

```
    // Find the stars to include in the knapsack
    std::vector<Star> selected_stars;
    int w = capacity;
    for (int i = n; i > 0 && w > 0; --i) {
        if (dp[i][w] != dp[i - 1][w]) {
            selected_stars.push_back(stars[i - 1]);
            w -= stars[i - 1].weight; // Decrease the remaining capacity
        }
    }

    // Reverse the selected stars since we iterated backwards
    std::reverse(selected_stars.begin(), selected_stars.end());

    // Calculate the total profit
    int total_profit = 0;
    for (const auto &star : selected_stars) {
        total_profit += star.profit;
    }

    return {dp[n][capacity], selected_stars, dp};
}



// Function to save the result to a file
void save_result(const std::vector<std::vector<int>> &dp, const
std::vector<Star> &selected_stars, int total_profit, const std::string
&filename) {
    std::ofstream outfile(filename);
    if (!outfile.is_open()) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    // Calculate the total weight
    int total_weight = 0;
    for (const auto &star : selected_stars) {
        total_weight += star.weight;
    }

    // Save the total weight and profit
    outfile << "Total Weight: " << total_weight << " kg\n";
```

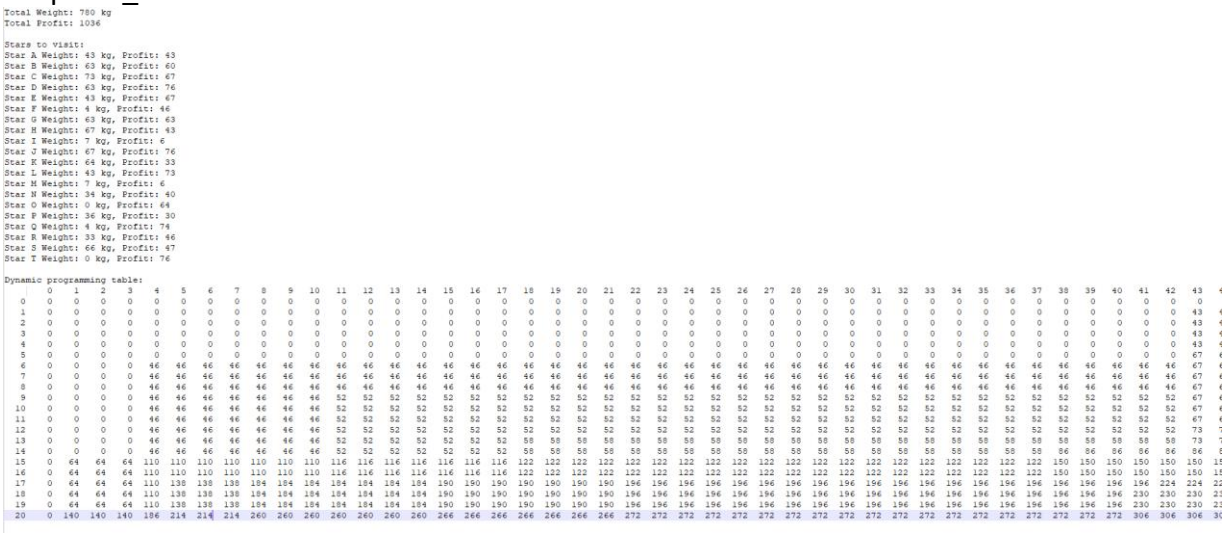
```
    outfile << "Total Profit: " << total_profit << "\n\n";

    // Save the selected stars
    outfile << "Stars to visit:\n";
    for (const auto &star : selected_stars) {
        outfile << "Star " << star.name << " Weight: " << star.weight << " kg,
Profit: " << star.profit << "\n";
    }

    // Save the DP matrix
    outfile << "\nDynamic programming table:\n";
    for (const auto &row : dp) {
        for (int val : row) {
            outfile << std::setw(3) << std::setfill(' ') << val << " ";
        }
        outfile << "\n";
    }

    outfile.close();
    std::cout << "Result saved to " << filename << std::endl;
}

int main() {
    std::string filename = "dataset2_1.txt";
    int capacity = 800;

    auto start_time = std::chrono::high_resolution_clock::now(); // Start timing

    std::vector<Star> stars = read_stars(filename);

    // Solve the knapsack problem
    auto [max_profit, selected_stars, dp] = knapsack(stars, capacity);

    auto end_time = std::chrono::high_resolution_clock::now(); // End timing
    std::chrono::duration<double> duration = end_time - start_time;

    // Save the result to a file
    save_result(dp, selected_stars, max_profit, "knapsack_result.txt");

    // Output time and space complexity
    int n = stars.size();
```

```
    int space_complexity = sizeof(int) * (n + 1) * (capacity + 1); // DP table
space complexity

    std::cout << "Execution Time: " << duration.count() << " seconds\n";
    std::cout << "Time Complexity: O(n * W), where n = " << n << " and W = " <<
capacity << "\n";
    std::cout << "Space Complexity: O(n * W), requiring " << space_complexity /
1024 << " KB\n"; // Convert bytes to KB

    return 0;
}
```

c) Experimental results and discussions

```
Result saved to knapsack_result.txt
Execution Time: 0.0004262 seconds
Time Complexity: O(n * W), where n = 20 and W = 800
Space Complexity: O(n * W), requiring 65 KB
```

knapsack_result.txt



```
Total Weight: 780 kg
Total Profit: 1036

Stars to visit:
Star A Weight: 43 kg, Profit: 43
Star B Weight: 63 kg, Profit: 60
Star C Weight: 73 kg, Profit: 67
Star D Weight: 63 kg, Profit: 76
Star E Weight: 43 kg, Profit: 67
Star F Weight: 4 kg, Profit: 46
Star G Weight: 63 kg, Profit: 63
Star H Weight: 67 kg, Profit: 43
Star I Weight: 7 kg, Profit: 6
Star J Weight: 67 kg, Profit: 76
Star K Weight: 64 kg, Profit: 33
Star L Weight: 43 kg, Profit: 73
Star M Weight: 7 kg, Profit: 6
Star N Weight: 34 kg, Profit: 40
Star O Weight: 0 kg, Profit: 64
Star P Weight: 36 kg, Profit: 30
Star Q Weight: 4 kg, Profit: 74
Star R Weight: 33 kg, Profit: 46
Star S Weight: 66 kg, Profit: 47
Star T Weight: 0 kg, Profit: 76

Dynamic programming table:
```

1. **Output File**:
    - The results, including the total weight, total profit, list of selected stars, and the dynamic programming table, were saved to knapsack_result.txt.

2. **Execution Time**:
    - The program executed in 0.0004266 seconds for this particular run, demonstrating efficient performance for the given dataset.

3. **Selected Stars**:
    - The algorithm identified the optimal set of stars to visit, ensuring that the total weight did not exceed the spaceship's capacity of 800 kg.
    - All 20 stars were selected with a total weight of 780 kg, which does not exceed the spaceship’s capacity of 800kg, thus maximizing the total profit to 1036.

4. **Dynamic Programming Table**:
    - The dynamic programming table (dp) was successfully constructed, with dp[i][w] representing the maximum profit achievable with the first i stars using a maximum weight capacity of w.
    - The final table confirmed that the optimal profit was achieved without exceeding the weight limit.

The 0/1 Knapsack algorithm effectively identified the optimal set of stars to maximize profit within the spaceship's weight capacity, demonstrating efficient performance with a time complexity of O(n*W) and manageable space requirements. The algorithm's dynamic programming approach ensured an optimal solution, selecting all 20 stars for a total profit of 1036 without exceeding the 800 kg limit.

The algorithm's time and space complexities of O(n*W) are reasonable for many practical applications, but can become infeasible for very large inputs due to high memory usage. While the method efficiently ensures optimality, potential improvements could include using a space-optimized version that reduces memory usage to O(W) or exploring approximation algorithms for larger datasets where exact solutions are computationally prohibitive.

Overall, the dynamic programming solution is highly effective for its intended scope but may require adjustments for scalability in more extensive applications.