CCP6214 Algorithm Design and Analysis

« Project Report »

Lecture Section: TC3L

Tutorial Section: T10L & T11L

Group Number: 9

Num	Student ID	Student Name	Task Descriptions	Percentage
1	241EC240NH	Morin Thomas (leader)	Q1	25
2	1211103146	Cheryl Gwee En Xin	Q4	25
3	1191202880	Abdullah Muawia Mustafa Hummeida	Q2	25
4	1211104262	Julian Vijay	Q3	25

Contents

1 Introduction

This report covers the analysis, implementation, and evaluation of algorithms applied to two datasets in three problem domains. It's divided into four sections: dataset generation, sorting algorithms, graph algorithms, and dynamic programming. Each section describes the algorithms used, provides step-by-step illustrations, presents experiment results, code snippets, discussions with reasoning, and concludes on the algorithms' effectiveness.

1. Dataset Generation

   This section shows how the dataset is created. It includes and explains the algorithms for generating Dataset 1 and Dataset 2, along with the output snippet for Dataset 1, and graph illustration for Dataset 2.

2. Sorting Algorithms

   In this section, Heap Sort and Selection Sort are applied to Dataset 1. The implementation is provided along with the part of the output. The comparison of timing vs dataset size is portrayed through a graph, and the time and space complexity of the algorithm is discussed.

3. Graph Algorithms

   Dijkstra's Algorithm and Kruskal's Algorithm are used to identify the shortest path and the Minimum Spanning Tree respectively. The illustration of the graph representing the output is drawn, and the space and time complexities are also discussed.

4. Dynamic Algorithms

   In this section, the 0/1 Knapsack Algorithm within the domain of dynamic programming is applied on Dataset 2. The resulting matrix is drafted, and the stars to visit, its weight and profit are all listed out. The space complexity is also discussed.

2 Question 1

2.1 Dataset 1

For dataset 1, the group leader's ID is used as the random seed reference to generate the dataset. The data must only contain unique digits that appear in the group leader's student ID. In this case, the group leader's student ID which is "241EC240NH", is modified to become "2419724082" by changing each letter within the ID to the last digit of the ASCII value.

To create our first datasets, we will use this function :

```
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <string>

std::vector<int> ContainsNumber(long long seed)
{
  std::vector<int> numbers(10, 0);
  while(seed != 0)
  {
    int tmp = seed%10;
    seed/=10;
    numbers.at(tmp) +=1;
  }

  std::vector<int> result;
  for(size_t i = 0; i < numbers.size(); i++)
  {
    if(numbers.at(i) == 0)
    {
      result.push_back(i);
    }
  }
  return result;
}

bool Contains(std::vector<int> numbersToAvoid, int number)
{
    while (number != 0)
  {
        int CurrentNumber = number % 10;
    for(size_t i = 0; i < numbersToAvoid.size(); i++)
    {
      if (CurrentNumber == numbersToAvoid.at(i))
      {
        return 0;
      }
    }
        number /= 10;
    }
  return 1;
}

void generateDataWithSeed(long long seed, int numPoints,
  const std::string& filename)
```

```
47 {
48   //Set the seed for the random number generator
49   std::mt19937 generator(seed);
50
51   //Generate random integer data points
52   std::uniform_int_distribution<int> distribution(0, 500);
53
54   std::ofstream file(filename);
55   //Create & open a file named filename for writing
56
57   if (file.is_open())
58   {
59     std::vector<int> NumberToAvoid = ContainsNumber(seed);
60
61     int i = 0;
62     while(i < numPoints)
63     {
64       int tmp = distribution(generator);
65
66       if(Contains(NumberToAvoid, tmp))
67       {
68         i++;
69         file << tmp << std::endl;
70       }
71     }
72
73     file.close();
74     std::cout << "Data written to " << filename << std::endl;
75   }
76   else
77   {
78     std::cerr << "Unable to open file: " << filename << std::endl;
79   }
80 }
```

This function will create our 6 datasets based on the seed which is the student ID of our group leader. For further explanation on how we may produce the dataset, let's break down the purpose of each line within the function to provide a clearer understanding on how it contributes to generating the dataset:

The line 49: an instance of the std::mt19937 class is created. This class is a pseudo-random number generator based on the Mersenne Twister. Seed is used to initialize the generator, meaning that each time we use this function with the same seed, we will get the same sequence of random numbers. This ensures reproducibility of results.

The line 52: an object of the std::uniform_int_distribution class is created to generate uniformly distributed integers in the range from 0 to 500. This object will be used to generate actual data values.

And after the line 64: where the data is generated and written to the file using the generator and the distribution specified before, and that all the digits contained in the number are present in the seed.

We chose to save our dataset in a specific text file to avoid having 6 arrays in the memory. However, this means that we will have to add functions that will transform our dataset in a file into data that we can manipulate, like an array or a heap.

The output will be 6 text files. The first twenty lines of the generated output will be shown as an example below:

```
22
424
298
482
417
297
29
217
180
140
11
449
111
484
211
401
222
209
280
447
```

2.2 Dataset 2

To begin, we will create a structure to represent each node in the graph. This struct, named "star" will encompass all the necessary parameters for each node, which will be utilized in the subsequent functions that will be implemented.
For each star, we will have its:

- Name (example: A, B, ..., T)
- Coordinates in a 3D space (i.e. x, y, z coordinates)
- Weight
- Profit

```
//Define a structure for representing a star
struct Star
{
  std::string name;
  int x, y, z;
  int weight;
  int profit;
};
```

The weight and profit will be used in the last function implemented.

A struct is needed to represent all 20 stars. In order to do so, we will use the following function that employs a seed. This function will generate 20 stars, each with a minimum of 3 edges:

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

#include "../../Include/DataSet1/dataSet1.hpp"

//Define a structure for representing a star
struct Star
{
  std::string name;
  int x, y, z;
  int weight;
  int profit;
};

std::vector<int> CreateNumbers(long long seed, std::vector<int> NumberToAvoid)
{
  std::mt19937 generator(seed);
  std::uniform_int_distribution<int> distribution(0, 500);

  std::vector<int> result;

  int i = 0;
  while(i != 100)
  {
```

```
    int tmp;
    do
    {
      tmp = distribution(generator);
    }while(!Contains(NumberToAvoid, tmp));

    result.push_back(tmp);
    i++;
  }

  return result;
}

//Function to generate a dataset of stars and routes
void generateDatasetWithSeed(std::vector<Star>& stars, std::vector<std::pair<Star,
  Star>>& routes, long long seed)
{
  std::mt19937 generator(seed);

  std::vector<int> NumberToAvoid = ContainsNumber(seed);

  std::vector<int> randomNumbers = CreateNumbers(seed, NumberToAvoid);

  //Generate 20 stars
  for(int i = 0; i < 20; i++)
  {
    char tmp = i+65;
    std::string name;
    name += tmp;

    int k = 5*i;
    int x = randomNumbers[k];
    int y = randomNumbers[k+1];
    int z = randomNumbers[k+2];
    int weight = randomNumbers[k+3];
    int profit = randomNumbers[k+4];
    stars.push_back({name, x, y, z, weight, profit});
  }

  //Connect each star to at least 3 other stars to create 54 routes
  std::vector<Star> allstar;
  for(const auto& star : stars)
  {
    allstar.push_back(star);
  }
  for(const auto& star : stars)
  {
    std::shuffle(allstar.begin(), allstar.end(), generator);
    for(int i = 0; i < 3; ++i)
    {
      if(star.name != allstar[i].name)
        routes.push_back({star, allstar[i]});
      else
        routes.push_back({star, allstar[19-i]});
    }
  }
}
```

Additionally, we have developed other functions that are capable of parsing the dataset into a text file and DOT file, and a function that can create an image representing the graph from the DOT file using Graphviz. The generated image will be named "Universe.png" and visually depicts the 20 stars and their respective edges.

```
//Function to write the dataset in a text file
void writeDatasetToFile(const std::vector<Star>& stars,
  const std::vector<std::pair<Star, Star>>& routes, std::string filename)
{
  std::ofstream file(filename);

  if (file.is_open())
  {
    file << "Star Name\t X\t Y\t Z\tWeight\tProfit\n";
    for (const auto& star : stars)
    {
      file << "\t" <<star.name << "\t\t" << star.x << "\t" << star.y << "\t"
        << star.z << "\t  " << star.weight << "\t " << star.profit << "\n";
    }

    file << "\nRoutes:\n";

    for (const auto& route : routes)
    {
      file << route.first.name << " -> " << route.second.name << "\n";
    }

    file.close();
    std::cout << "Dataset written to " << filename << "\n";
  }
  else
  {
    std::cerr << "Unable to open file\n";
  }
}
```

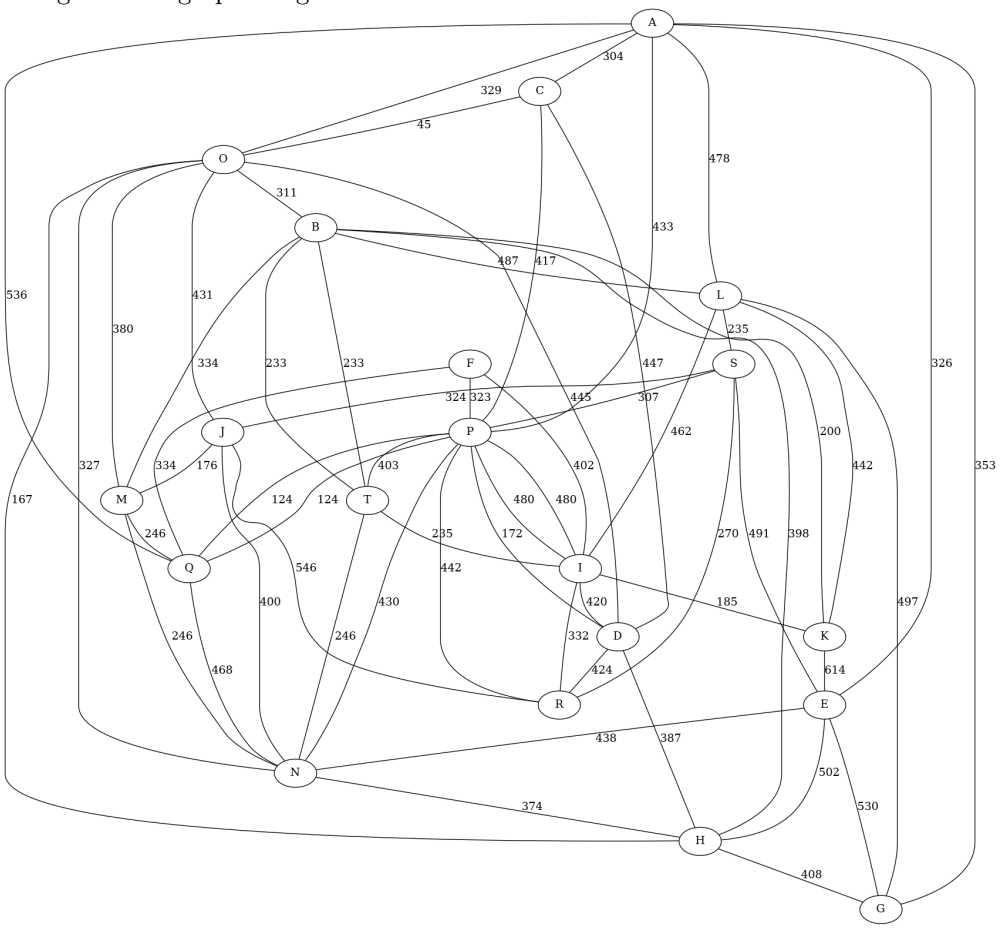
```
//Function to write the dataset in a .dot file
void createUndirectedGraphDotFile(const std::vector<std::pair<Star, Star>>& routes,
  std::string filename)
{
  std::ofstream file(filename);

  if(file.is_open())
  {
    file << "graph G {\n";
    for (const auto& route : routes)
    {
      double distance = sqrt(pow(double(route.second.x - route.first.x), 2.0)
        + pow(double(route.second.y - route.first.y), 2.0)
          + pow(double(route.second.z - route.first.z), 2.0));
      file << "\t" << route.first.name << " -- " << route.second.name
        << " [label=\"" << static_cast<int>(distance) << "\"];\n";
    }
    file << "}\n";
    file.close();
    std::cout << "Undirected graph DOT file created: " << filename << std::endl;
  }
  else
```

```
  {
    std::cerr << "Unable to create undirected graph DOT file." << std::endl;
    filename.clear(); //Return an empty string if file creation fails
  }
}
```

```
//Function to generate a graph image from a DOT file using Graphviz
void generateGraphImage(const std::string& dotFile, std::string filename)
{
  std::string command = "dot -Tpng " + dotFile + " -o " + filename;
  int result = std::system(command.c_str());

  if (result == 0)
  {
    std::cout << "Undirected graph image created: " << filename << std::endl;
  }
  else
  {
    std::cerr << "Failed to generate undirected graph image." << std::endl;
  }
}
```

In order to generate all the necessary graphs, we will utilize Graphviz, an open-source graph visualization software. This enables us to streamline and automate the process by utilizing the dot files generated from the function mentioned earlier. Using these dot files, we can create images in the .png file format to visualize our graphs.

The generated graph image from the DOT file:



We have also obtained a text file which contains all the stars and their coordinates:

```
Star Name  X   Y   Z  Weight  Profit
  A   437 361 33    170  46
  B   144 460 130   6  170
  C   416 477 314   16   103
  D   331 61  174   404  343
  E   410 36  43    371  310
  F   371 361 467   41   146
  G   117 467 140   460  44
  H   364 341 440   431  436
  I   137 433 147   13   406
  J   144 137 474   113  440
  K   73  471 317   64   430
  L   331 133 440   67   160
  M   146 173 301   103  36
  N   136 363 144   360  407
  O   377 473 337   443  117
  P   374 63  341   161  374
  Q   300 36  437   344  430
  R   467 463 177   341  411
  S   367 363 407   73   360
  T   371 430 174   171  0
```

In this file, the details regarding all the vertices for each star has also been noted down. For brevity, only a few of them will be shown below:

```
Routes:
A -> C
A -> O
A -> P
B -> L
B -> T
B -> M
C -> O
C -> P
C -> D
D -> R
D -> P
```

3 Question 2

3.1 Heap Sort

For the HeapSort functions, we have taken the function used in a tutorial session [1]. But first we need to create the "PriorityQueue" based on the dataset, so for that we need to iterate over the file to create it. This is what the code below does.

```
PriorityQueue<int> CreateHeapFromDatabase(const string& filename)
{
  ifstream file(filename);
  PriorityQueue<int> result;

  if(file.is_open())
  {
    string line;
    while(getline(file, line))
    {
      if(!line.empty() && line.back() == '\n')
      {
        line.pop_back();
      }
      try
      {
        int num = stoi(line);
        result.enqueue(num);
      }
      catch(const invalid_argument&)
      {
        cerr << "Invalid number format: " << line << endl;
      }
    }
    file.close();
  }
  else
  {
    cerr << "Unable to open file: " << filename << endl;
  }
  return result;
}
```

Then, we have the code taken from the tutorial session. We haven't changed anything. We just added a function to dequeue the whole heap:

```
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <string>
#include <cstdio>
#include <stdexcept>

using namespace std;

template <typename T>
class PriorityQueue
{
```

```
14  vector<T> A;
15
16  //used in heapify enqueue.
17  void heapify_enqueue(size_t index)
18  {
19    if (index == 0) // if already at root.
20      return;
21
22    // parent index
23    size_t p_index = (index-1)/2;
24
25    // swap if parent is smaller
26    if (A[p_index] < A[index])
27    {
28      swap(A[p_index], A[index]);
29      // recursion of the function
30      heapify_enqueue(p_index);
31    }
32  }
33
34  void heapify_dequeue(int index)
35  {
36    size_t greatest = index; // max index
37
38    // left child index
39    size_t left_c = 2*index+1;
40
41    // right child index
42    size_t right_c = 2*index+2;
43
44    // compare and find the greatest child
45    if(left_c < A.size() && A[left_c] > A[greatest])
46    {
47      greatest = left_c;
48    }
49    if (right_c < A.size() && A[right_c] > A[greatest])
50    {
51      greatest = right_c;
52    }
53
54    if (greatest != index)
55    {
56      swap(A[index], A[greatest]);
57      heapify_dequeue(greatest); // recursion
58    }
59  }
60
61  public:
62  void enqueue(T element)
63  {
64    A.push_back(element);
65    heapify_enqueue(A.size()-1); // start at last element.
66  }
67
68  T dequeue()
69  {
70    if(A.empty())
71    {
```

```
      throw out_of_range("Heap is empty");
    }
    T removed_element = A[0];
    A[0] = A[A.size()-1];
    A.pop_back();
    if(!A.empty())
    {
      heapify_dequeue(0);  // 0 is a valid size_t value
    }
    return removed_element;
  }

  size_t size()
  {
    return A.size();
  }

  int get(int index)
  {
    if (index >= A.size())
    {
      throw out_of_range("Index out of range");
    }
    return A.at(index);
  }
};

void dequeueAll(PriorityQueue<int> heap)
{
  while(heap.size() > 0)
  {
    heap.dequeue();
  }
}
```

And here, the function to write our dataset sorted in a text file:

```
void writeDatasetToFile(PriorityQueue<int> heap, const string& filename)
{
  ofstream file(filename);

  if(file.is_open())
  {
    for(size_t i = 0; i < heap.size(); i++)
    {
      file << heap.get(i) << endl;
    }
    file.close();
    cout << "Data written to " << filename << "\n";
  }
  else
  {
    cerr << "Unable to open file\n";
  }
}
```

The output is saved in a text file, we will display the start and the end of the first HeapSet, so there are only 100 numbers into it:

```
497
494
482
484
494
448
449
422
447
474
```

```
101
11
110
108
187
7
70
44
11
104
```

Time Complexity: HeapSort is a sorting method known to be highly efficient and it belongs to the comparison sorting techniques. It works on the principle that it organizes the stream of input data into a heap structure; in turn, it extracts the largest element from a heap and reconstructs the heap, and this process continues until all of the elements are sorted. The time complexity of HeapSort is as follows:The time complexity of HeapSort is as follows:

Best Case: O(nlogn) Average Case: O(nlogn) Worst Case: O(nlogn)

In all the cases, HeapSort is able to perform O(nlogn) operations due to the fact that the construction of the heap is O(n) and the heapification Selecting n elements from the heap requires O(logn) time where n elements are representing the size of the heap. The logarithmic factor is because the binary heap in the tree is of log n height.

Space Complexity: HeapSort Sort has a space complexity of O(1). This is because HeapSort is an in-place sorting algorithm, this means that HeapSort will just use a few additional variables and memory space other than the input array needed to execute the sorting algorithm. The sorting is done such that the program utilizes space that is at most constant with respect to the input size. Another advantage of HeapSort is its memory usage; because of the ways in which the heaps that are used are arranged, this sort method is especially convenient in situations where as much memory as possible needs to be saved.

3.2 Selection Sort

Like HeapSort, we need to browse the dataset that we will have to use to sort it. So we create an array in the form of a vector with all the numbers in the dataset.

```
vector<int> CreateArrayFromDatabase(const string& filename)
{
  ifstream file(filename);
  vector<int> result;

  if(file.is_open())
  {
    string line;
    while(getline(file, line))
    {
      try
      {
        if(!line.empty() && line.back() == '\n')
        {
          line.pop_back();
        }
        int num = stoi(line);
        result.push_back(num);
      }
      catch(const invalid_argument&) //Added exception handling
      {
        cerr << "Invalid number format: " << line << endl;
      }
    }
    file.close();
  }
  else
  {
    cerr << "Unable to open file: " << filename << endl;
  }
  return result;
}
```

After creating our vector from our dataset, we can sort our vector with the selectionSort's function below. We took this function from a tutorial session [1], and adapted it to best fit our goal. Sorting by selection (of the maximum) in an array of numbers of size n consists of going through it several times and placing the largest element at the end, then the 2nd smallest element next to last, then the 3rd smallest element in its place, etc. Sorting by selection is done in place.

```
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <string>
#include <cstdio>
#include <stdexcept>

using namespace std;

void selectionSort(vector<int> &array)
{
  int n = array.size();
  for(int i = n-1; i > 0; i--)
```

```
  {
    int maxIndex = i;
    for(int j = 0; j < i; j++)
    {
      if(array[j] > array[maxIndex])
      {
        maxIndex = j;
      }
    }

    if(maxIndex != i)
    {
      swap(array[i], array[maxIndex]);
    }
  }
}
```

Thus, just like we did in the heapSort file, we need to save our dataset sorted somewhere. So we wrote this function to save our array in a text file.

```
void writeDatasetToFile(vector<int> array, const std::string& filename)
{
  ofstream file(filename);

  if(file.is_open())
  {
    for(const auto& elem : array)
    {
      file << elem << endl;
    }
    file.close();
    cout << "Data written to " << filename << "\n";
  }
  else
  {
    cerr << "Unable to open file\n";
  }
}
```

Now that the output is saved in a text file, we will display the start and the end of the first set of selection sort, so that there are only 100 numbers into it:

```
1
2
2
7
7
11
11
11
17
21
```

```
449
474
474
482
484
492
```

```
492
494
494
497
```

Time Complexity: The sorting technique that is dealt with here is Selection Sort, which is one of the simplest comparison sorting techniques. It operates by iteratively selecting the minimum element from the portion of the array that is as yet unsorted and then swapping this with the leftmost element in the array that is part of the unsorted section. The time complexity of Selection Sort is as follows:The time complexity of Selection Sort is as follows: Best Case: O(n2) Average Case: O(n2) Worst Case: O(n2)

The number of comparisons is also O(n2) at any circumstances because Selection Sort goes through the list to find the smallest element in the unsolved subarray portion and put it into the right position within the sorted portion. This indicates that the time required for executing Selection Sort algorithm is of quadratic nature and hence the algorithm's efficiency decreases when the size of the given dataset increases, which is not good for large datasets.

Space Complexity: The space complexity of Selection Sort is Constant or O(1) as it does not require more space. Similar to HeapSort, Selection Sort also helps in sorting the elements within the specified given space. It scans the array and sorts it and the best thing is that it does so without much memory required, which is proportional to the input size. It requires only a fixed additional space to accommodate a variable or two like 'index of the minimum element' and a 'temporary swap space'.

4 Question 3

4.1 Shortest Path

Now, we need to write a function to find the shortest path from A to the other star. We start by initializing the dimensions of our matrix, because since we will browse it in some functions, having a variable somewhere would be useful.

```
//Matrix size
#define rows 20
#define columns 20
```

But in order to traverse our graph, we must create an adjacency matrix with the distances between each "star" that we will use when executing the main function (Dijkstra). To create this adjacency matrix, we will browse our vector/array with all the paths between each star and put the distance between each of them in the matrix. And if there is not path between a star and another, the distance will be -1.

```
//Create Adjacency matrix from Universe (Graph with 20 stars)
vector<vector<int>> CreateAdjacencyMatrix(vector<pair<Star, Star>> routes)
{
  vector<vector<int>> Adj_M(rows, vector<int>(columns, -1));

  int tmp_Star1;
  int tmp_Star2;

  for(size_t i = 0; i < routes.size(); i++)
  {
    //Compute the distance between each star
    Star star1 = routes[i].first;
    Star star2 = routes[i].second;
    double distance_d = sqrt(pow(double(star2.x - star1.x), 2.0)
       + pow(double(star2.y - star1.y), 2.0)
        + pow(double(star2.z - star1.z), 2.0));

    //Add the distance in the adjacency matrix
    tmp_Star1 = static_cast<int>(star1.name[0]) - 65;
    tmp_Star2 = static_cast<int>(star2.name[0]) - 65;

    int distance_i = static_cast<int>(distance_d);

    Adj_M[tmp_Star1][tmp_Star2] = distance_i;
    Adj_M[tmp_Star2][tmp_Star1] = distance_i;
  }

  return Adj_M;
}
```

The function below will be used to find the shortest distance between all possible paths. This function will be called in the main function.

```
int minDistance(vector<int> dist, vector<bool> M)
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int j = 0; j < rows; j++)
    if (M[j] == false && dist[j] <= min)
      min = dist[j], min_index = j;

  return min_index;
}
```

Here, we have the main function, which will take as parameters the adjacency matrix found before and the star from where we will start.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <limits.h>

using namespace std;

//Function to execute Dijkstra Algorithm
pair<vector<int>, vector<vector<int>>> Dijkstra(
    const vector<vector<int>>& Adj_M, int start)
{
    vector<bool> M(rows, false);
    vector<int> dist(rows, INT_MAX);
    vector<vector<int>> paths(rows);

    dist[start] = 0;

    for(int i = 0; i < rows-1; i++)
    {
        int min_dist = minDistance(dist, M);
        M[min_dist] = true;

        for(int j = 0; j < rows; j++)
        {
            if(!M[j] && Adj_M[min_dist][j] != -1
                && dist[min_dist] != INT_MAX
                && dist[min_dist] + Adj_M[min_dist][j] < dist[j])
            {
                dist[j] = dist[min_dist] + Adj_M[min_dist][j];
                paths[j] = paths[min_dist];
                paths[j].push_back(min_dist);
            }
        }
    }

    return make_pair(dist, paths);
}
```

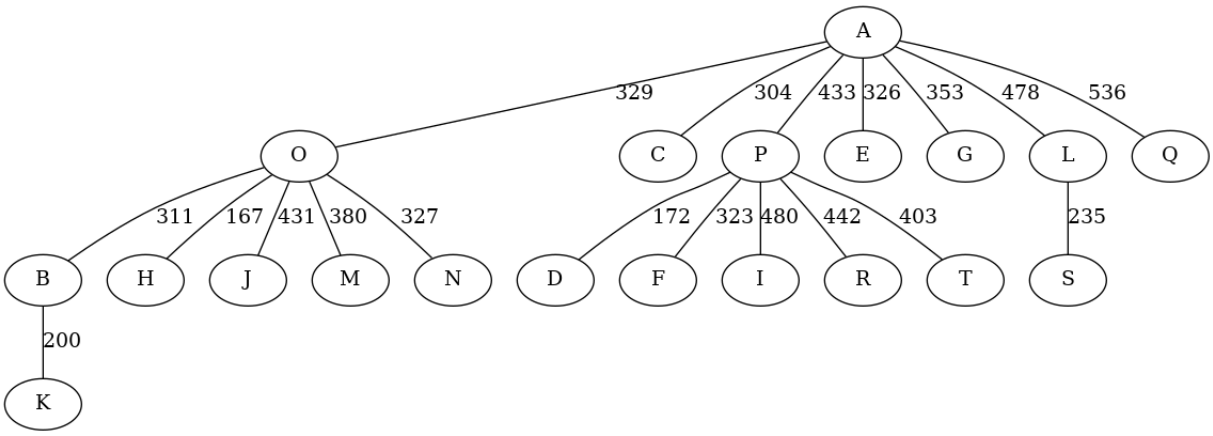
So this function creates and does a lot of things at the same time, like calculate the shortest path and save it to all star in the graph. So we will explain what the function does. Firstly, we create three vectors which are : a vector where we will store which star we have visited or not, another one which will store the shortest distance between each stars and finally a vector of vector (matrix), that will store the path from the start to all the others stars. Thanks to this last vector, we will be able to create a new graph with all the shortest paths from the start to the others stars. After this three lines, we set the distance from A to zero (from A to A there is a distance of 0) so we don't have to search for this. We create a loop to iterate on all the stars but not the last one because this one is already in the shortest path. In this loop, we call the function minDistance which will find the non-visited stars with the current shortest distance among all stars. We will name this star : "current" which is min_dist in the function. We mark current as visited, and we iterate over all the stars adjacent to current. After that, for each iteration, we need to know if the star has not been already visited, if we can reach it from "current", if the current distance is not INT_MAX and if the path found is shorter than the path found previously. Thus, if all these conditions are true, we can update the minimal distance from current to the star 'j'. And, finally, we can update the shortest path from the start to the star 'j' as well as add current at the end of the path.

Now that we have all the shortest paths from the start to the other stars, we need to create a text file with the source, the destination and their distance as an output. And with that we will generate an image from a ".dot" file. These two functions are displayed below, but the one that generates the image is in the dataset2 part.

```
//Function to write all shortest paths in a text file
void CreateTxtFileWithDistance(vector<int> distance)
{
  string filename = "Output/Dijkstra/shortest_paths.txt";

  //Open the .txt file for writing
  ofstream txtFile(filename);
  if (!txtFile)
  {
    cerr << "Error: Unable to create .txt file." << endl;
    exit(EXIT_FAILURE);
  }

  txtFile << "Src\tdest\tShortest Distance" << endl;

  for(size_t i = 0; i < distance.size(); i++)
  {
    txtFile << " A\t " << static_cast<char>(i + 65)
      << "\t\t\t\t" << distance[i] << endl;
  }

  txtFile.close();
  cout << "Data written to " << filename << endl;
}
```

```cpp
//Function to build the .dot file from Dijkstra's result
void CreateDotFile(const vector<int>& dist,
  const vector<vector<int>>& paths, const string& fileName,
    const vector<vector<int>>& Adj_M)
{
  ofstream dotFile(fileName);
  if (!dotFile)
  {
    cerr<< "Error: Unable to create .dot file." <<endl;
    exit(EXIT_FAILURE);
  }

  dotFile << "graph shortest_paths {" << endl;

  for(int i = 0; i < rows; i++)
  {
    if(dist[i] != INT_MAX)
    {
      for(size_t j = 0; j < paths[i].size(); j++)
      {
        if(Adj_M[paths[i][j]][i] != -1)
          dotFile << "\t" << static_cast<char>(paths[i][j] + 65)
            << " -- " << static_cast<char>(i + 65)
              << " [label=\"" << Adj_M[paths[i][j]][i]
                << "\"];" << endl;
      }
    }
  }

  dotFile << "}" << endl;

  dotFile.close();
}
```

The output of the Dijkstra algorithm is:



And here is the text file filled with the source, destination and distance:

```
Src dest  Shortest Distance
 A   A         0
 A   B         640
 A   C         304
 A   D         605
 A   E         326
 A   F         756
 A   G         353
 A   H         496
 A   I         913
 A   J         760
 A   K         840
 A   L         478
 A   M         709
 A   N         656
 A   O         329
 A   P         433
 A   Q         536
 A   R         875
 A   S         713
 A   T         836
```

The principal factors that affect time complexity are the utilization of an adjacency matrix and the use of a linear search for determining the smallest distance vertex. The distance array and visited set are initialized at the beginning in O(V) time. Here, V designates the number of vertices. The minDistance function conducts a linear search along the distance array to locate the vertex with the smallest distance that has not yet been visited; this operation takes O(V) time. Our search took V times to execute so the time complexity of finding the nearest vertex would be $O(V^2)$.

In addition, the program may relax the edges of each node, examining all of its neighbors to modify their distances. The worst-case scenario involves making V checks for each of the V vertices leading to an $O(V^2)$ time complexity for the edge relaxation step. In this implementation, the overall time complexity of Dijkstra's algorithm is $O(V^2)$ when the developer combines these steps.

Storage requirements of the adjacency matrix drive the space complexity. The adjacency matrix, which stores the weights of the edges between each two vertices, needs $O(V^2)$ memory. Hence, the program's space complexity leans towards the adjacency matrix, therefore $O(V^2)$.

4.2 Minimum Spanning Tree

To identify the minimum spanning tree, we need to implement the Kraskal algorithm. At the start, we have to create two struct : 'Edges' and 'Subset', 'Edges' will contain all the vertices/edges from the graph and Subset will be a part of the minimum spanning tree like a cloud in class.

```
//Structure to represent an edge
struct Edge {
  int src, dest, distance;
};

//Structure to represent a subset for Union-Find
struct Subset {
  int parent, rank;
};
```

So for all the edges we need to declare from where they come from, to where they go and the distance between theses two points. For the subset struct we have two parameters. The first variable is one of the stars present in the subset and the second is the number of stars in the subset.

Now that we have the struct, we need to create all of the edges, because we have the paths between each star but we need to have access to their distances in order to be able to sort the edges in order to have a good complexity.

So, this is the code to create all edges present in the graph.

```
vector<Edge> CreateAllEdges(vector<pair<Star, Star>> routes)
{
  vector<Edge> edges;

  for(size_t i = 0; i < routes.size(); i++)
  {
    //Compute the distance between each star
    Star star1 = routes[i].first;
    Star star2 = routes[i].second;
    double distance = sqrt(pow(double(star2.x - star1.x), 2.0)
      + pow(double(star2.y - star1.y), 2.0)
        + pow(double(star2.z - star1.z), 2.0));

    Edge edge;
    edge.src = static_cast<int>(star1.name[0])-65;
    edge.dest = static_cast<int>(star2.name[0])-65;
    edge.distance = static_cast<int>(distance);

    edges.push_back(edge);
  }

  return edges;
}
```

In order to have a good complexity, we need to sort all the edges present in our graph in an non-decreasing order. To do that, we use a selection sort to sort this vector because since there are not a lot of data, whether we take a good sort or not won't be significant.

```
void EdgesSort(vector<Edge> &edges)
{
  int n = edges.size();

  for(int i = n-1; i > 0; i--)
  {
    int maxIndex = i;
    for(int j = 0; j < i; j++)
    {
      if(edges[j].distance >
        edges[maxIndex].distance)
      {
        maxIndex = j;
      }
    }

    if(maxIndex != i)
    {
      swap(edges[i], edges[maxIndex]);
    }
  }
}
```

The 'Find' function finds the subset to which a particular element belongs. It takes two parameters: 'subsets', which is a vector storing subsets information, and 'i', which is the index of the element whose subset needs to be found. It recursively finds the ultimate parent of the subset using path compression, where each subset's parent is updated to the root parent during the recursive call to optimize future find operations. Finally, it returns the index of the subset's parent.

```
//Find operation for Union-Find
int find(vector<Subset>& subsets, int i)
{
  if(subsets[i].parent != i)
  {
    subsets[i].parent = find(subsets, subsets[i].parent);
  }
  return subsets[i].parent;
}
```

The 'Union' function performs the union of two subsets based on their ranks. It takes three parameters: 'subsets', 'x', and 'y', where 'x' and 'y' are the indices of the elements whose subsets need to be united. It first finds the root parent of the subsets to which 'x' and 'y' belong. Then, based on the ranks of the subsets, it decides which subset should be the parent of the other. If both subsets have the same rank, one subset is chosen as the parent, and its rank is incremented. This operation optimizes the time complexity of the algorithm.

```
//Union operation for Union-Find
void Union(vector<Subset>& subsets, int x, int y)
{
  int xroot = find(subsets, x);
  int yroot = find(subsets, y);

  if(subsets[xroot].rank < subsets[yroot].rank)
  {
    subsets[xroot].parent = yroot;
  }
  else if(subsets[xroot].rank > subsets[yroot].rank)
  {
    subsets[yroot].parent = xroot;
  }
  else
  {
    subsets[yroot].parent = xroot;
    subsets[xroot].rank++;
  }
}
```

The 'Kruskal' function takes a vector of edges ('edges') and the number of vertices ('V') as input and returns the Minimum Spanning Tree (MST) of the graph represented by these edges. It initializes an empty vector MST to store the edges of the MST and creates 'V' subsets, each containing one vertex. Then, it sorts the input edges in non-decreasing order based on their weights. Next, it iterates through the sorted edges and selects each edge one by one. For each selected edge, it checks if adding it to the MST creates a cycle or not. If adding the edge does not create a cycle, it adds the edge to the MST and performs the Union operation on the subsets of the edge's source and destination vertices. This process continues until 'V-1' edges have been added to the MST or all edges have been considered. Finally, it returns the MST.

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

using namespace std;

vector<Edge> KruskalMST(vector<Edge>& edges, int V)
{
  int n = edges.size();
  vector<Edge> MST;
  vector<Subset> subsets(V);

  for(int i = 0; i < V; i++)
  {
    subsets[i].parent = i;
    subsets[i].rank = 0;
  }

  EdgesSort(edges);

  int e = 0, i = 0;
  while(e < V-1 && i < n)
  {
```

```
    Edge next_edge = edges[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    if(x != y)
    {
      MST.push_back(next_edge);
      Union(subsets, x, y);
      e++;
    }
  }

  return MST;
}
```

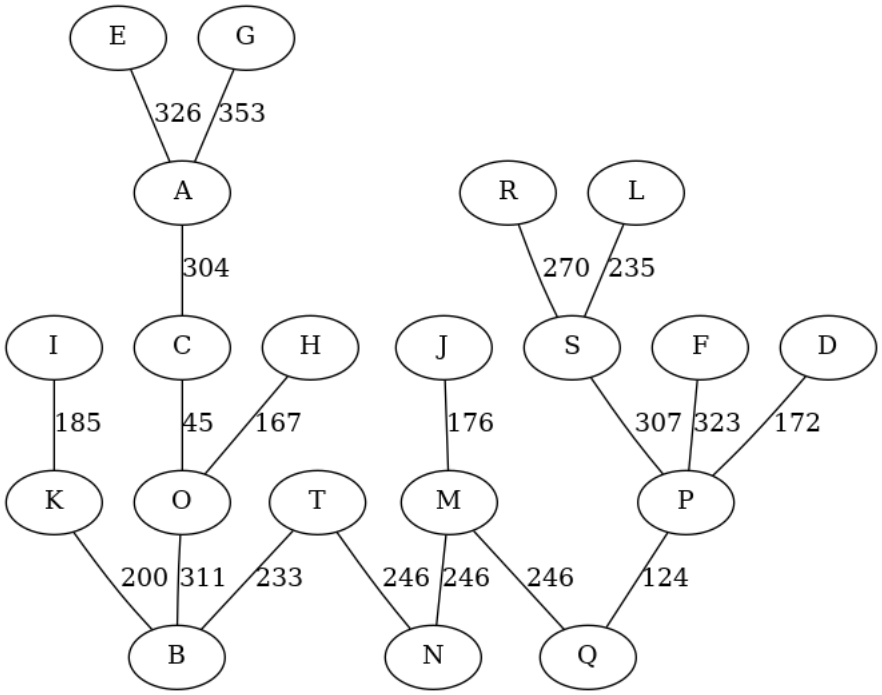
So for the output of this function we need to display a graph that represents our MST, and also a text file with all the edges present in the MST.

```
void CreateTxtFileMST(vector<Edge>& MST)
{
  string filename = "Output/Kruskal/MST.txt";

  //Open the .txt file for writing
  ofstream txtFile(filename);
  if (!txtFile)
  {
    cerr<< "Error: Unable to create .txt file."<< endl;
    exit(EXIT_FAILURE);
  }

  txtFile << "Src\tdest\tDistance" << endl;

  for(const Edge& edge : MST)
  {
    txtFile << static_cast<char>(edge.src + 65) << " "
            << static_cast<char>(edge.dest + 65) << "\t\t\t"
                << edge.distance << endl;
  }

  txtFile.close();
  cout << "Data written to " << filename << endl;
}
```

```
void convertToDot(const vector<Edge>& MST, string filename)
{
  ofstream dotFile(filename);

  if(dotFile.is_open())
  {
    dotFile << "graph MST {" << endl;
    for(const Edge& edge : MST)
    {
      dotFile << "\t" << static_cast<char>(edge.src+65)
        << " -- " << static_cast<char>(edge.dest+65)
          << " [label=\"" << edge.distance << "\"];"
            << endl;
    }
```

```
    dotFile << "}\n";
    dotFile.close();

    cout << "The .dot file of the minimum spanning tree "
      << "has been successfully created." << endl;
  }
  else
  {
    cerr << "Error: Unable to open .dot file." << endl;
    filename.clear();
  }
}
```

The output of the Kruskal algorithm is:



And there is also a text file where there are all the vertices saved from the Minimum Spanning Tree (MST).

```
Src dest  Distance
C O     45
P Q     124
H O     167
D P     172
J M     176
I K     185
K B     200
T B     233
L S     235
M N     246
M Q     246
T N     246
R S     270
A C     304
S P     307
O B     311
F P     323
E A     326
G A     353
```

The algorithm sorts these edges using selection sort, a method that has a time complexity of $O(E^2)$. Once the sorting of the edges is done, the algorithm makes use of union-find operations to prevent cycles from occurring during the addition of edges to the MST. In order to optimize the union-find operations, the algorithm uses path compression and union by rank. These tweaks guarantee that nearly constant time is dedicated to each operation. The total time complexity of this exact implementation is $O(E^2)$. The space requirements are O(E+V) comprising the edges storage, the Union-Find structure, and the MST itself.

5 Question 4

5.1 Dynamic Programming / Knapsack

To identify the optimal set of stars that we can conquer with our ship that has a capacity of 800kg, we will employ the 0/1 Knapsack Algorithm. The approach taken implements a dynamic programming approach to solve the knapsack problem. The result will consist of the list of stars that are the optimal starts to conquer and plunder. For this problem, we have tried to take a loose reference from another project [2] to better understand and develop the solution.

We first implement the knapsack algorithm by creating and filling in a 2D array which calculates the best profit for each planet as the capacity increases to maximum:

```
vector<vector<int>> generate2DArray(vector<Star> stars, int capacity)
{
  int numStars = stars.size();  //Number of stars

  // Create a 2D vector to store results of subproblems.
    // dp[i][w] will store maximum profit that can be obtained
  // with weight limit w and using only the first i stars
  vector<vector<int>> dp(numStars+1, vector<int>(capacity+1, 0));

  // Fill the dp array using bottom-up dynamic programming
  for (int i = 1; i <= numStars; ++i) //for each row (star)
  {
    int currentProfit = stars[i-1].profit;
    int currentWeight = stars[i-1].weight;

    for (int j = 1; j <= capacity; ++j) // for each column (capacity)
    {
      if (stars[i-1].weight <= j) // if star can be included (i.e. weight less than
    or equal to capacity)
      {
        // Choose the maximum value between including it and excluding it
        dp[i][j] = max(currentProfit + dp[i-1][j-currentWeight], dp[i-1][j]);
      }
      else //If the current star cannot be included, then just exclude it
      {
        dp[i][j] = dp[i-1][j];
      }
    }
  }

  return dp;
}
```

This function takes a vector of stars ('stars') and an integer representing the knapsack capacity ('capacity') as input. It creates a 2D array 'dp' to store the results of subproblems, where 'dp[i][w]' represents the maximum profit achievable with a weight limit of 'w' and using only the first 'i' stars. The array is instantiated as a 2D vector, where the number of rows are: $the number of stars + 1$, and the number of columns are: the $capacity + 1$, and it is initialized with the value of 0 for each element. It iterates over each star and each possible weight limit, calculating the maximum profit that can be obtained by either including or excluding the current star. If the current star's weight exceeds the current weight limit, it simply copies the maximum profit obtained without considering the current star from the previous row.

Below is the main code to find the optimal list of planets to plunder through the 2D array that we have already generated:

```
vector<Star> Knapsack(vector<Star> stars, vector<vector<int>> dp, int capacity)
{
  int numStars = stars.size();  // Number of stars

  // Retrieve the chosen stars from the dp array
  int result = dp[numStars][capacity];   // Start at last cell which is maximum
    profit
  int remWeight = capacity;              // Remaining weight
  vector<Star> chosenStars;              // Vector to store chosen stars

  for (int i = numStars; i > 0 ; --i) // For each star
  {
    if (result != dp[i-1][remWeight]) // Check if star was chosen by comparing to
    previous (if not, do nothing)
    {
      // Add it to the list of chosen stars
      chosenStars.push_back(stars[i-1]);

      // Update remaining profit and weight
      result -= stars[i-1].profit;
      remWeight -= stars[i-1].weight;
    }
  }

  //Reverse the order of chosen stars since we iterated backwards
  reverse(chosenStars.begin(), chosenStars.end());

  return chosenStars;
}
```

After filling the 'dp' array, the function traces back through the array to determine which stars were chosen to maximize the total profit. It starts from the bottom-right corner of the 'dp' array and iterates backward, checking whether including the current star would increase the total profit. If including the current star results in a higher profit, it adds the star to the list of chosen stars and updates the remaining profit and weight accordingly. This process continues until the entire 'dp' array is traversed or until the remaining profit becomes zero. Finally, it reverses the order of the chosen stars since they were added in reverse order during the tracing process. It returns a vector containing the stars that should be placed into the knapsack to maximize the total profit without exceeding the capacity.

To display the resulting matrix of the 2D array, we use this function that exports to a .csv file as the spreadsheet-like format makes it slightly easier to navigate the resulting record:

```
void CreateCSVFileMatrix(vector<vector<int>>& dp, string filename) {
  ofstream outputFile(filename);

  if (outputFile.is_open())
  {
    for (int i = 1; i < dp.size(); ++i)
    {
      for (int j = 1; j < dp[i].size(); ++j) {
        outputFile << dp[i][j] << ',';
      }
```

```
      outputFile << '\n';
    }
    outputFile.close();
    cout << "Data written to " << filename << endl;
  }
  else
  {
    cerr << "Unable to open file for writing.";
  }
}
```

For the output of the resulting algorithm, we have a csv file as follows. The columns (representing the capacity, kg) that are displayed here are columns 75 - 84:

0	0	0	0	0	0	0	0	0	0
170	170	170	170	170	170	170	170	170	170
273	273	273	273	273	273	273	273	273	273
273	273	273	273	273	273	273	273	273	273
273	273	273	273	273	273	273	273	273	273
419	419	419	419	419	419	419	419	419	419
419	419	419	419	419	419	419	419	419	419
419	419	419	419	419	419	419	419	419	419
722	825	825	825	825	825	825	825	825	825
722	825	825	825	825	825	825	825	825	825
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006
722	825	836	836	836	836	836	836	1006	1006

To display all the stars that a ship with 800kg of capacity can take, we created a function to write in a text file all the stars taken by this ship:

```
void CreateTxtFileChosenStars(vector<Star>& chosenStars,
  string filename)
{
  //Write output to file
  ofstream outputFile(filename); //Open a file for writing

  if (outputFile.is_open())
  {
    //Write chosen stars along with their
    weights and profits to the file
    for (const auto& star : chosenStars)
    {
```

```
      outputFile << star.name << " Weight: "
        << star.weight << " Profit: "
          << star.profit << "\n";
    }
    outputFile.close(); //Close the file
    cout << "Data written to " << filename << endl;
  }
  else
  {
    cerr << "Unable to open file for writing.";
    //Error message if file cannot be opened
  }
}
```

For the output of the entire Knapsack algorithm, we have a text file:

```
B Weight: 6 Profit: 170
C Weight: 16 Profit: 103
I Weight: 13 Profit: 406
J Weight: 113 Profit: 440
K Weight: 64 Profit: 430
P Weight: 161 Profit: 374
Q Weight: 344 Profit: 430
S Weight: 73 Profit: 360
```

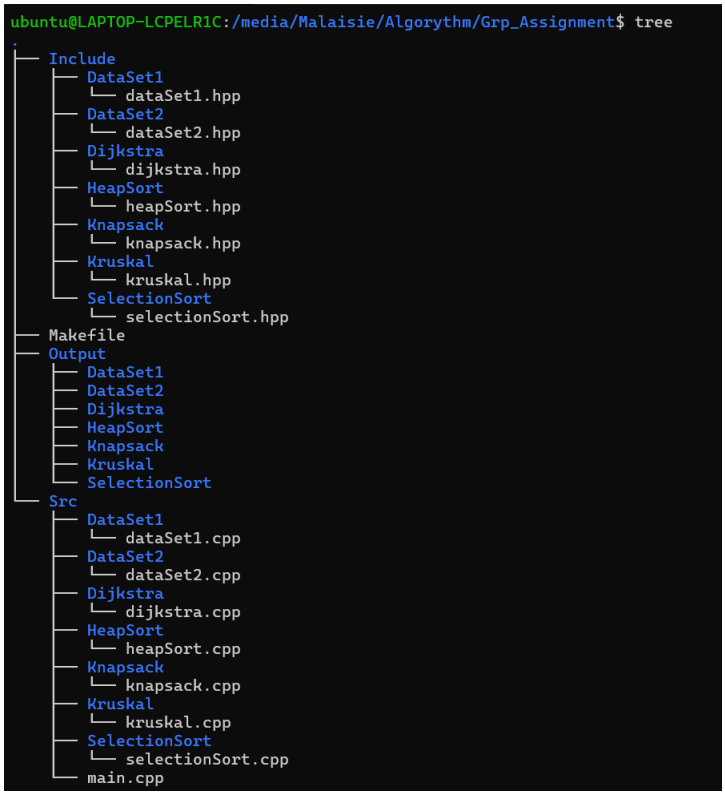
By taking the bottom-up approach for the 0/1 Knapsack problem instead of the recursion approach, we use a function to solve the subproblems of the knapsack problem, storing these results in a 2D array to avoid redundant computations. Through this method, we may effectively reduce the time complexity for the algorithm as each subproblem is only solved once. There are $N8W$ unique states, where N represents the number of items and W represents the capacity. This is because each item can either be included or excluded, and the capacity of the knapsack varies from 0 up to W. As each state takes a constant time of O(1) to calculate, we may effectively reduce the time complexity from O(2^N) (for recursive approach) to O($N * W$).

The space complexity is determined by the size of the 2D array. The dimensions of the table is $(N + 1) * (W + 1)$, where the extra row and column is useful for handling base cases without requiring strict conditional checks and for making indexing simple and straightforward. Thus, the use of space is reduced to O($N * W$) as that is the size for the 2D array used for storing results of subproblems.

6 Submission

6.1 Directory Hierarchy

The directory hierarchy for our program is structured as depicted in the image below:



We have 3 folders:

- The first, with all the source codes where the previously written functions are stored in their respective folders, so generally all the files ending with ".cpp". At the root level, there is a final integration point of our project—the main.cpp file which initializes the program, executes the necessary operations, and orchestrates the flow of control by calling other functions and modules.

- The second is a folder where all the header files ending with ".hpp" are located. These files will be used for compilation only and will also be used to link some cpp files together.

- The last folder will be where the output of all the functions will be stored. This includes .png, .dot, .txt, and .csv files.

And with all these 3 folders, we have a file named "Makefile" which will facilitate the compilation across all files locted in these different folders.

6.2 Use

So to compile all the code, we need to go the root of the project, write this little command :

```
$ make
```

After that, all the code will compile and we will have a file named 'main'. It's our file with which we can execute the project. We have two ways to execute the project. The first one is to write the command below, which will execute the project with the ID already saved into the file 'main.cpp', so it will be easier for us to not have to write our student ID each time we execute the project.

```
$ ./main
```

The other way is to specify all the IDs by writing this command below. It is possible to change the '4' by a '3' because a group can be formed by 3 or 4 people. After this command, you will need to write the ID of the leader and then the others. When all the IDs have been written, the project will execute itself. And after 15-20 minutes (because of the selection sort) we can see all the output in the folder 'Output'.

```
$ ./main 4
```

To clean the project we have implemented two methods, the first one is below, and allows you to remove all the object files, the main file and the output file made.

```
$ make clean
```

And the other one, it's a command that will allow you to remove only the main file and the object files, and let the output files as they are.

```
$ make clean-ObjectFile
```

References

[1] Multimedia University Lecturers. *Lab 5 Priority Queues and Heaps [lab manual]*. 2024.

[2] W. Fiset. *Algorithms*. https://github.com/williamfiset/Algorithms. 2017.