Assignment Report | AED

Teachers:
Tomás Oliveira e Silva
João Manuel Rodrigues

The Assignment Problem

Hugo Paiva, 93195
João Laranjo, 91153
Lucas Sousa, 93019

DETI
Universidade de Aveiro

13-11-2019

Contents

1 Introduction

Inserted on the curricular plan of the course Algorithms and Data Structures, this report is a result of the code of "The Assignment Problem", proposed by the teachers.

This is an approach on how n agents (a) can be assigned to n tasks (t) such that the total cost (**C**a,t) of assignment is minimized (or even maximized). The matrix of costs is randomly generated, depending on a seed, thus the minimum/maximum cost can change with every execution.

1.1 Getting Started

These instructions will help to compile and run developed programs on your local machine. All the code can be found in a GitHub repository. A clone can be made using the next command, if you have permissions[1]:

```
git clone https://github.com/hugofpaiva/AED_P1
```

1.2 Prerequisites

To compile programs, it is necessary to have a C compiler like cc installed on your local machine.

1.3 Compiling

The following command compiles the assignment program (assignment.c) where `<executable_filename>` will be the executable filename:

```
cc -Wall -O2 assignment.c -o <executable_filename> -lm
```

1.4 Running

Options:

```
-e ..................... Uses Brute Force method for n=3 and n=5;

-f <seed> ................ Uses Brute Force method with specified
                           Seed value;

-b <seed> ................ Uses Branch and Bound method with specified
                           Seed value;

-r <seed> <seed> <seed> <N> Uses Brute Force method with Random
                           Permutations generated N times and three
                           specified Seed values;

-g <seed> ................ Uses Greedy method with specified Seed value;

-a <seed>................. Runs all the previous methods with specified
Seed value.
```

[1] For confidentiality reasons, the repository may be private.

2 Defined Functions

This functions are used throughout some methods of implementation.

2.1 costAssignment

The 'costAssignment' function inhere the calculation of the passed permutation cost as well as adding one occurrence to the permutation cost on the histogram.

Later on, the 'histogram[permutation_cost]' will be used to render the histogram, and the 'permutation_cost' will be necessary to the The Random Permutation, Brute Force , and The Branch-and-Bound methods.

Let:

- (int) 'n' be the number of agents/tasks;
- (int) 'permutation_cost' be a temporary variable of cost relative to the actual permutation. This value is updated in every iteration of the for loop;
- (static int) 'histogram[i]' be the array that holds the number of occurrences relative to 'i', the 'permutation_cost'.

```
int costAssignment(int n, int a[n]){
  int permutation_cost = 0;

  for (int i = 0; i < n; i++){
    permutation_cost += cost[i][a[i]];
  }
  histogram[permutation_cost]++;
  return permutation_cost;
}
```

2.2 init_costs

The 'init_costs' function lies on the generation of the matrix constituted by the costs assigned to a certain agent (a) and task (t). During the generation of this matrix, the function also has the responsibility to calculate the 'min_int_cost' and the 'max_init_cost', both the minimum and maximum cost of all costs. To do so, this variables were started as a very large integer and a very small integer[2], respectively, making afterwards a comparison during the creation of a cost of the matrix of costs. This will determine if the actual generated cost is smaller or larger than this variables.

Later on, the 'min_init_cost' and the 'max_init_cost' will be helpful on The Branch-and-Bound method.

Let:

- (static int) 'cost[a][t]' be cost relative to agent 'a' and the task 't';
- (static int) 'min_int_cost', 'max_int_cost' be the variables that hold the minimum and maximum cost of all costs.

```
static void init_costs(int n){
...
   assert(n >= 1 && n <= max_n);
  srandom((unsigned int)seed * (unsigned int)max_n + (unsigned int)n);
  min_init_cost = plus_inf;
  max_init_cost = minus_inf;
  for (int a = 0; a < n; a++)
    for (int t = 0; t < n; t++)
    {
      cost[a][t] = 3 + (random() % range) + (random() % range) + (random() % range); // [3,3*range]
      if (cost[a][t] < min_init_cost)
        min_init_cost = cost[a][t];
      if (cost[a][t] > max_init_cost)
        max_init_cost = cost[a][t];
    }
}
```

[2]Defined previously.

3 Methods of Implementation

3.1 Brute Force Method

The Brute Force method consists in computing all the permutations of agents and tasks and for each permutation calculate the estimated cost through the matrix of costs.

This method is capable of calculating not only the minimum and maximum cost of all the permutations relative to 'n' agents and tasks, but also the permutation associated with that minimum and maximum cost.

The function receives the number of agents/task, the index of the permutation and a permutation. Then it swap between the various values of the permutation and when the index of the permutation reaches the end of the permutation (n), the cost of that permutation is computed. If it is the potential minimum or maximum cost, it stores the actual permutation.

Despite being a simple method to implement, the Brute Force search is a very exhaustive method, thus it's time consuming.

The computational complexity of this method is $\mathcal{O}(n \times n!)$.

Let:

- (int) 'n' be the number of agents/tasks;
- (int) 'm' be the index of the actual permutation;
- (int) 'a[n]' be the given permutation;
- (int) 'permutation_cost' be a temporary variable of the cost. This value is updated every iteration of the for loop when 'm' = 'n';
- (static int) 'min_cost', 'max_cost' be the variables that hold the minimum and max cost value of all permutations;
- (static int) 'min_cost_assignment', 'max_cost_assignment' be the variables that hold the permutation associated with minimum and max cost value of all permutations, respectively;

```
static void generate_all_permutations(int n, int m, int a[n])
{
  if (m < n - 1)
  {
    for (int i = m; i < n; i++)
    {
#define swap(i, j)
  do
  {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
  } while (0)
      swap(i, m);
      generate_all_permutations(n, m + 1, a);
      swap(i, m);
#undef swap
```

```
    }
  }
  else
  {
    int permutation_cost = costAssignment(n, a);
    if (permutation_cost > max_cost)
    {
      max_cost = permutation_cost;
      for (int i = 0; i < n; i++)
        max_cost_assignment[i] = a[i];
    }
    if (permutation_cost < min_cost)
    {
      min_cost = permutation_cost;
      for (int i = 0; i < n; i++)
        min_cost_assignment[i] = a[i];
    }
    n_visited++;
  }
}
```

3.2 The Random Permutation Method

The Random Permutation method consists in computing a large[3] number of permutations of size 'n' and calculating the minimum and maximum cost for those computed permutations.

Once the number of different permutations far exceeds the computed number of permutations[4], the minimum and maximum cost are going to be an approximation of the real costs because not all possibilities are able to be computed.

To reach this goal, the function is reused various times, in each starting by receiving the number of agents/task and a permutation. After that, the permutation is randomized and the cost of that permutation is computed as well as the confirmation if it is the potential minimum or maximum cost, storing the actual permutation, when confirmed.

This method is typically faster than the Brute-Force method[5].

Let:

- (int) '`n`' be the number of agents/tasks;
- (int) '`t[n]`' be the given permutation;
- (int) '`permutation_cost`' be a temporary variable of the cost.
- (static int) '`min_cost`', '`max_cost`' be the variables that hold the minimum and max cost value of all permutations;
- (static int) '`min_cost_assignment`', '`max_cost_assignment`' be the variables that hold the permutation associated with minimum and max cost value of all permutations;

```
void random_permutation(int n, int t[n])
{
  assert(n >= 1 && n <= 1000000);
  for (int i = 0; i < n; i++)
    t[i] = i;
  for (int i = n - 1; i > 0; i--)
  {
    int j = (int)floor((double)(i + 1) * (double)random() / (1.0 +
    (double) RAND_MAX));
    assert(j >= 0 && j <= i);
    int k = t[i];
    t[i] = t[j];
    t[j] = k;
  }
  int permutation_cost = costAssignment(n, t);
  if (permutation_cost > max_cost)
  {
    max_cost = permutation_cost;
    for (int i = 0; i < n; i++)
    {
      max_cost_assignment[i] = t[i];
    }
  }
  if (permutation_cost < min_cost)
  {
```

[3] By a 'large' it's meant a number like 1e6;
[4] E.g. 10! > 1e6;
[5] Depends on the number of permutations computed as well as the size 'n'.

```
26      min_cost = permutation_cost;
27      for (int i = 0; i < n; i++)
28        min_cost_assignment[i] = t[i];
29    }
30  }
```

3.3 The Branch-and-Bound Method

3.3.1 The Branch-and-Bound Method for minimum value

The Branch-and-Bound method for minimum value consists in computing the cost of a permutation of n agents and if it's cost is bigger than the minimum cost of a previous permutation, it immediately discards the current permutation and moves on to the next one.

The function receives the number of agents/tasks, the index of the permutation, a permutation and the partial cost of the permutation until the index 'm'.

This is accomplished by checking every time the function is called, if the actual minimum cost is smaller than the smallest possible minimum cost relative to the rest of the permutation.

If not, the permutation is swapped between the various values of the permutation and when the index of the permutation (m) reaches the end of the permutation (n), the cost of that permutation is computed, as well as the confirmation if it's the potential minimum cost, storing the actual permutation, when confirmed.

If the actual minimum cost is smaller than the smallest possible minimum cost relative to the rest of the permutation, it discards the current permutation and moves on to the next one because it is know the best value.

Let:

- (int) 'n' be the number of agents/tasks;
- (int) 'm' be the index of the actual permutation;
- (int) 'a[n]' be the given permutation;
- (int) 'partial_cost' be the partial cost of the permutation until the index 'm' ;
- (int) 'permutation_cost' be a temporary variable of the cost. This value is updated every iteration of the for loop when 'm' = 'n';
- (static int) 'min_int_cost' be the variable that hold the minimum cost of all costs;
- (static int) 'min_cost' be the variable that hold the minimum cost value of all permutations;
- (static int) 'min_cost_assignment' be the variable that hold the permutation associated with minimum cost value of all permutations.

Version: Minimum Cost

```
static void generate_all_permutations_branch_and_bound_min(int n, int m,
int a[n], int partial_cost)
{
  if (min_cost < (min_init_cost * (n - m) + partial_cost))
  {
    return;
  }
  else
  {
    if (m < n - 1)
    {
      for (int i = m; i < n; i++)
      {
```

```
#define swap(i, j)
  do
  {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
  } while (0)
        swap(i, m);
        generate_all_permutations_branch_and_bound_min(n, m + 1, a,
        (partial_cost + cost[m][a[m]]));
        swap(i, m);
#undef swap
      }
    }
    else
    {
      int permutation_cost = costAssignment(n, a);
      if (permutation_cost < min_cost)
      {
        min_cost = permutation_cost;
        for (int i = 0; i < n; i++)
        {
          min_cost_assignment[i] = a[i];
        }
      }
      n_visited++;
    }
  }
}


```

3.3.2 The Branch-and-Bound Method for maximum value

The Branch-and-Bound method for maximum value consists in computing the cost of a permutation of n agents and if its cost is smaller than the maximum cost of a previous permutation it immediately discards the current permutation and moves on to the next one.

The function receives the number of agents/tasks, the index of the permutation, a permutation and the the partial cost of the permutation until the index 'm'.

This is accomplished by checking every time the function is called, if the actual maximum cost is bigger than the biggest possible maximum cost relative to the rest of the permutation.

If not, the permutation is swapped between the various values of the permutation and when the index of the permutation (m) reaches the end of the permutation (n), the cost of that permutation is computed, as well as the confirmation if it's the potential maximum cost, storing the actual permutation, when confirmed.

If the actual maximum cost is bigger than the biggest possible maximum cost relative to the rest of the permutation, it discards the current permutation and moves on to the next one because it is know the best value.

Let:

- (int) 'n' be the number of agents/tasks;
- (int) 'm' be the index of the actual permutation;
- (int) 'a[n]' be the given permutation;
- (int) 'partial_cost' be the partial cost of the permutation until the index 'm' ;
- (int) 'permutation_cost' be a temporary variable of the cost. This value is updated every iteration of the for loop when 'm' = 'n';
- (static int) 'max_int_cost' be the variable that hold the maximum cost of all costs;
- (static int) 'max_cost' be the variable that hold the maximum cost value of all permutations;
- (static int) 'max_cost_assignment' be the variable that hold the permutation associated with the maximum cost value of all permutations.

Version: Maximum Cost

```
static void generate_all_permutations_branch_and_bound_max(int n, int m,
int a[n], int partial_cost)
{
  if (max_cost > (max_init_cost * (n - m) + partial_cost))
    return;
  else
  {
    if (m < n - 1)
    {
      for (int i = m; i < n; i++)
      {
#define swap(i, j)
  do
  {
    int t = a[i];
```

```
    a[i] = a[j];
    a[j] = t;
  } while (0)
        swap(i, m);
        generate_all_permutations_branch_and_bound_max(n, m + 1, a,
        (partial_cost + cost[m][a[m]]));
        swap(i, m);
#undef swap
      }
    }
    else
    {
      int permutation_cost = costAssignment(n, a);
      if (permutation_cost > max_cost)
      {
        max_cost = permutation_cost;
        for (int i = 0; i < n; i++)
          max_cost_assignment[i] = a[i];
      }
      n_visited++;
    }
  }
}

```

3.4 The Greedy Method

3.4.1 The Greedy Method for minimum value

The Greedy Method for minimum value consists in iterating over the lines of a matrix of costs and finding the column which holds the minimum cost of that line. When that column is found, it's eliminated and so is the line, therefore that column and line won't be used again for the rest of the method. The estimated minimum cost of the matrix $n_{x}n$ is the sum of the minimum cost available for each line.

The minimum cost is an approximated value (almost always) because when we choose a column of a line we deny access to the rest of the elements on that column. This can lead to loss of the true minimum cost of a full line, making the value an approximation.

The permutation of the estimated minimum cost is calculated by getting the index of the various minimums costs corresponded to all lines. The line index will be the permutation array index and the column index will be the value of that permutation index.

The computational complexity of this method is $\mathcal{O}(n^2)$.

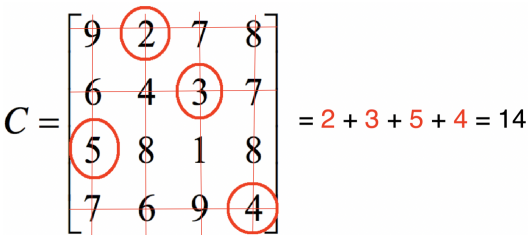

Figure 1: Example using a randomly-generated matrix C

Let:

- (int) '`n`' be the size of the matrix of costs;
- (int) '`binary_array`' be the array of ones and zeroes that enables me to use/not use a column of the matrix;
- (int) '`final_min_cost`' be the variable that hold the minimum cost through the Greedy Method;
- (int) '`c_pos`' be the variable that holds the position of the column to block in every iteration;
- (int) '`tmp_min_cost`' be the variable that is used for the first comparasion of numbers in each new line;
- (static int) '`min_cost_assignment`' be the variable that hold the permutation associated with the minimum cost value of all permutations.

Version: Minimum Cost

```
static void greedy_method_min(int n)
{
  // Declaration of the binary array that holds the possibility of using a column (0) or not (1)
  int binary_array[n];
  memset(binary_array, 0, n * sizeof(int));

  int final_min_cost = 0; // variable that holds the value of the cost using the Greedy Method (and Brute Force
    for the last k lines)
```

```
8
9   for (int l = 0; l < n; l++) // line
10  {
11    int c_pos; // holds the position of the column that has the minimum cost. It
12    is used to update 'binary_array'
13    int tmp_min_cost = plus_inf;
14
15    for (int c = 0; c < n; c++) // column
16    {
17      if (cost[l][c] <= tmp_min_cost && binary_array[c] == 0)
18      {
19        tmp_min_cost = cost[l][c];
20        c_pos = c;
21      }
22    }
23    binary_array[c_pos] = 1;
24    min_cost_assignment[l] = c_pos;
25    final_min_cost += tmp_min_cost;
26  }
27  min_cost = final_min_cost;
28 }
29
```

3.4.2 The Greedy Method for maximum value

The Greedy Method for maximum value consists in iterating over the lines of a matrix of costs and finding the column which holds the maximum cost of that line. When that column is found it's eliminated and so is the line, therefore that column and line won't be used again for the rest of the method. The estimated maximum cost of the matrix $n_x n$ is the sum of the maximum cost available for each line.

The maximum cost is an approximated value (almost always) because when we choose a column of a line we deny access to the rest of the elements on that column. This can lead to a loss of the true maximum cost of a full line, making the value an approximation .

The permutation of the estimated maximum cost is calculated by getting the index of the various maximums costs corresponded to all lines. The line index will be the permutation array index and the column index will be the value of that permutation index.

The computational complexity of this method is $\mathcal{O}(n^2)$.

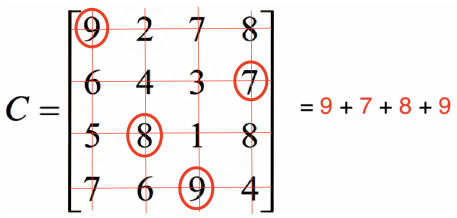

Figure 2: Example using a randomly-generated matrix C

Let:

- (int) '`n`' be the size of the matrix of costs;
- (int) '`binary_array`' be the array of ones and zeroes that enables me to use/not use a column of the matrix;
- (int) '`final_max_cost`' be the variable that hold the maximum cost through the Greedy Method;
- (int) '`c_pos`' be the variable that holds the position of the column to block in every iteration;
- (int) '`tmp_max_cost`' be the variable that is used for the first comparasion of numbers in each new line;
- (static int) '`max_cost_assignment`' be the variable that hold the permutation associated with the maximum cost value of all permutations.

Version: Maximum Cost

```
1  static void greedy_method_max(int n)
2  {
3    // Declaration of the binary array that holds the possibility of using a column (0) or not (1)
4    int binary_array[n];
5    memset(binary_array, 0, n * sizeof(int));
6    int final_max_cost = 0; // variable that holds the value of the cost using the Greedy Method (and Brute Force
       for the last k lines)
7    for (int l = 0; l < n; l++) // line
8    {
9      int c_pos; // holds the position of the column that has the minimum cost. It is used to update 'binary_array'
10     int tmp_max_cost = minus_inf;
```

```
    for (int c = 0; c < n; c++) // column
    {
      if (cost[l][c] >= tmp_max_cost && binary_array[c] == 0)
      {
        tmp_max_cost = cost[l][c];
        c_pos = c;
      }
    }
    binary_array[c_pos] = 1;
    max_cost_assignment[l] = c_pos;
    final_max_cost += tmp_max_cost;
  }
  max_cost = final_max_cost;
}

```

4 Results

The results were transformed into graphics using a C script that generated the data and then processed by a MATLAB script.

C script:

```
if ((what_to_show & show_histogram) != 0)
  {
    //start with base filename
    char baseFilename[] = "data/result";
    char baseFilenameH[] = "data/histo_cost";

    //place to store final final name
    const int maxSize = 50;
    char filename[maxSize];
    char filename_h[maxSize];

    sprintf(filename, "%s_%d.txt", baseFilename, seed);
    sprintf(filename_h, "%s_%d.txt", baseFilenameH, n);

    FILE *f = fopen(filename, "a+");
    FILE *fh = fopen(filename_h, "a+");
    if (f == NULL || fh == NULL)
    {
      printf("Erro a abrir o ficheiro!\n");
      exit(1);
    }
    else
    {
      fprintf(f, "%d\t\t%d\t\t%ld\t\t%.6f\t\t\t%d\t\t\t%d\n", seed, n, n_visited, cpu_time, min_cost, max_cost);
      for (int i = 0; i < max_n * t_range; i++)
      {
        fprintf(fh, "%d\n", histogram[i]);
      }
    }

    fclose(f);
    fclose(fh);

  }

```

Example of MATLAB script for the graphic about occurrences according to the costs, using the Method Generate All Permutations, with the seeds correspondent to the group students numbers and respectively Gaussian:

```
%% Generate All Permutations
% Seeds = 93195 93019 91153
clear;
clc;

T14_91153 = load("histo_cost_14_91153.txt");
T14_93195 = load("histo_cost_14_93195.txt");
T14_93019 = load("histo_cost_14_93019.txt");

n_costs = T14_91153;
[lin, col] = size(n_costs);
costs = [1 : 1 : lin];

c = costs';
```

```
h = n_costs;
m = sum(c.*h)/sum(h);
v = sum((c-m).^2.*h)/sum(h);
p = 1/sum(h);
d = sqrt(2)*erfinv(2*p);

first1 = find(h, 1, "first");
last1 = find(h, 1, 'last');

plot_1 = plot(c, h, "b", "LineWidth", 2)
hold on
plot_2 = plot(c, exp(-(c-m).^2/(2*v))./sqrt(2*pi*v).*sum(h), "r", "LineWidth", 1)
hold on
plot_3 = plot([c(first1) c(last1)], [h(first1) h(last1)], "bo", "LineWidth", 2)
hold on

title('GAP Group Seeds')
xlabel('Costs')
ylabel('Uses nOccurrences')
xlim([first1 last1])

n_costs = T14_93195;
[lin, col] = size(n_costs);
costs = [1 : 1 : lin];

c = costs';
h = n_costs;
m = sum(c.*h)/sum(h);
v = sum((c-m).^2.*h)/sum(h);
p = 1/sum(h);
d = sqrt(2)*erfinv(2*p);

first2 = find(h, 1, "first")
last2 = find(h, 1, 'last')

if first2 < first1
    first = first2
else
    first = first1
end

if last2 > last1
    last = last2
else
    last = last1
end

plot_4 = plot(c, h, "g", "LineWidth", 2)
hold on
plot_5 = plot(c, exp(-(c-m).^2/(2*v))./sqrt(2*pi*v).*sum(h), "r", "LineWidth", 1)
hold on
plot_6 = plot([c(first2) c(last2)], [h(first2) h(last2)], "go", "LineWidth", 2)
hold on

title('GAP Group Seeds')
xlabel('Costs')
ylabel('Uses nOccurrences')
xlim([first last])

n_costs = T14_93019;
[lin, col] = size(n_costs);
costs = [1 : 1 : lin];
```

```
77
78 c = costs';
79 h = n_costs;
80 m = sum(c.*h)/sum(h);
81 v = sum((c-m).^2.*h)/sum(h);
82 p = 1/sum(h);
83 d = sqrt(2)*erfinv(2*p);
84
85 first3 = find(h, 1, "first")
86 last3 = find(h, 1, 'last')
87
88 if first3 < first
89     first = first3
90 end
91
92 if last3 > last
93     last = last3
94 end
95
96 plot_7 = plot(c, h, "black", "LineWidth", 2)
97 hold on
98 plot_8 = plot(c, exp(-(c-m).^2/(2*v))./sqrt(2*pi*v).*sum(h), "r", "LineWidth", 1)
99 hold on
100 plot_9 = plot([c(first3) c(last3)], [h(first3) h(last3)], "blacko", "LineWidth", 2)
101 hold on
102
103 legend([plot_1 plot_2, plot_3, plot_4 plot_5, plot_6, plot_7, plot_8, plot_9], "91153", "Gauss", "Min/Max",
      "93195", "Gauss", "Min/Max", "93019", "Gauss", "Min/Max")
104
105 title('GAP Group Seeds')
106 xlabel('Costs')
107 ylabel('Uses nOccurrences')
108 xlim([first last])
109
```

4.1 Results from Generating All Permutations

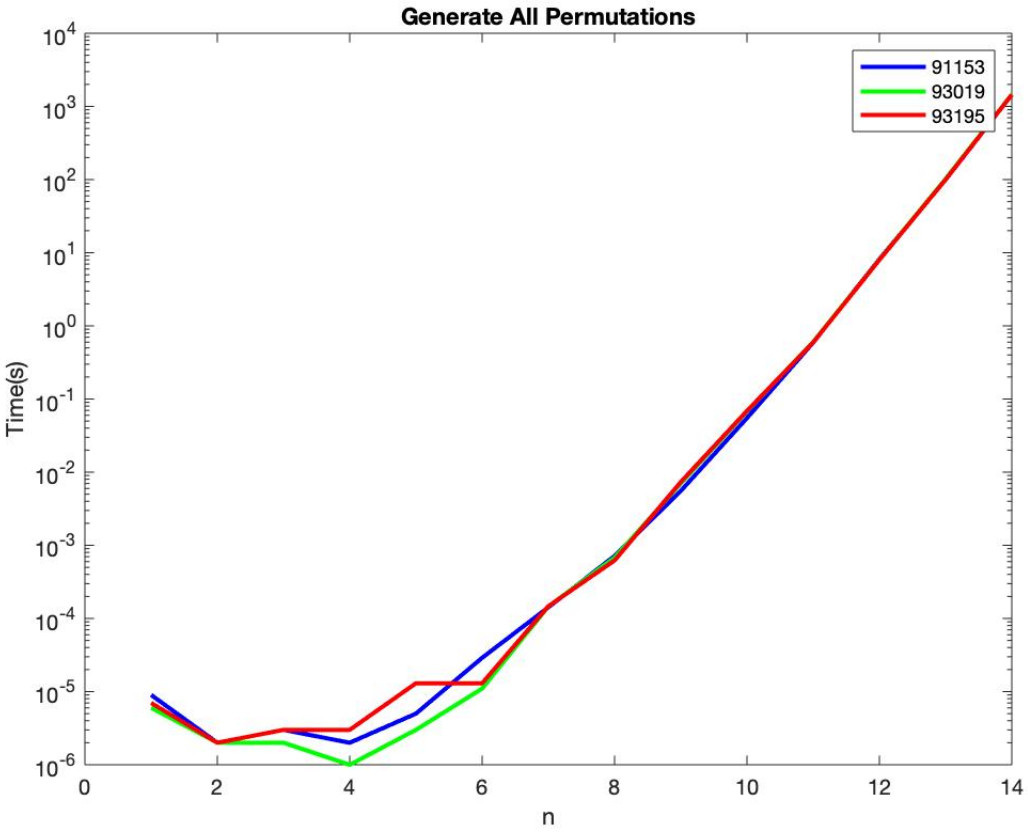

Figure 3: Time of execution for all seeds using the Generate All Permutations method

It is observed that for n < 10, the execution time is nearly 0 (solution is almost instant) but, as it is raised the n, it increases rapidly.

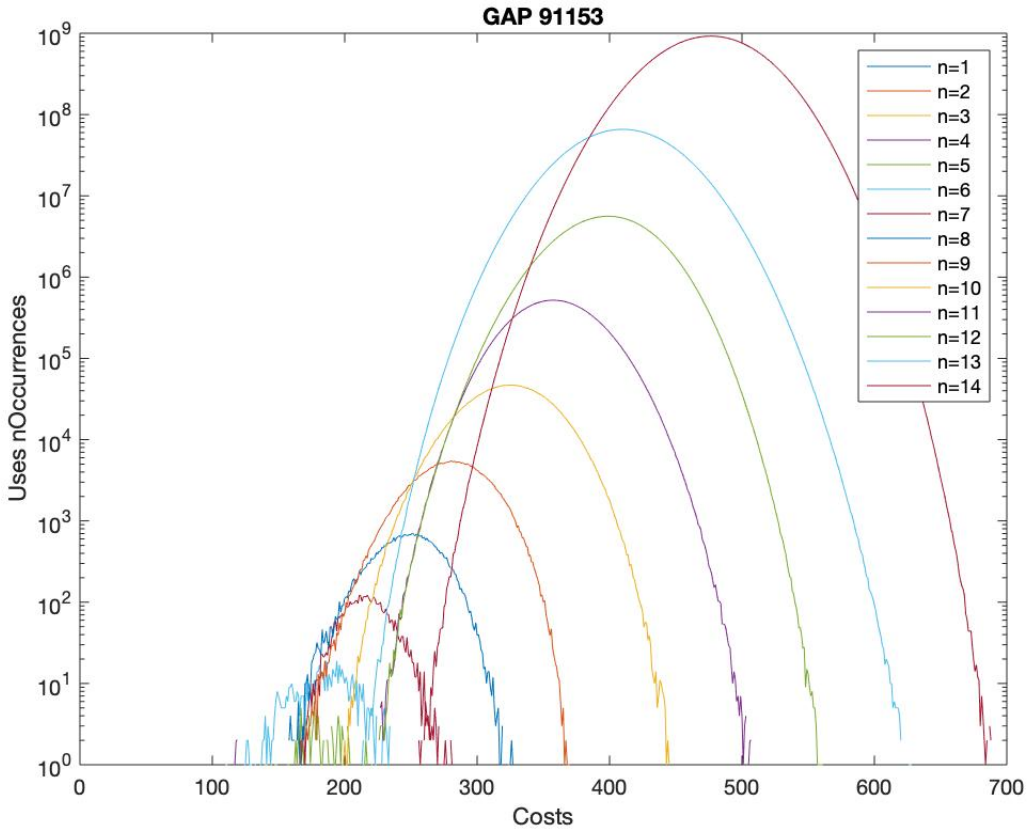

Figure 4: Occurrences according to the permutations costs, for each n, using the Generate All Permutations method with seed 91153

It is observed that with a larger n, the minimum and maximum raises and the graph tends to form a better curve

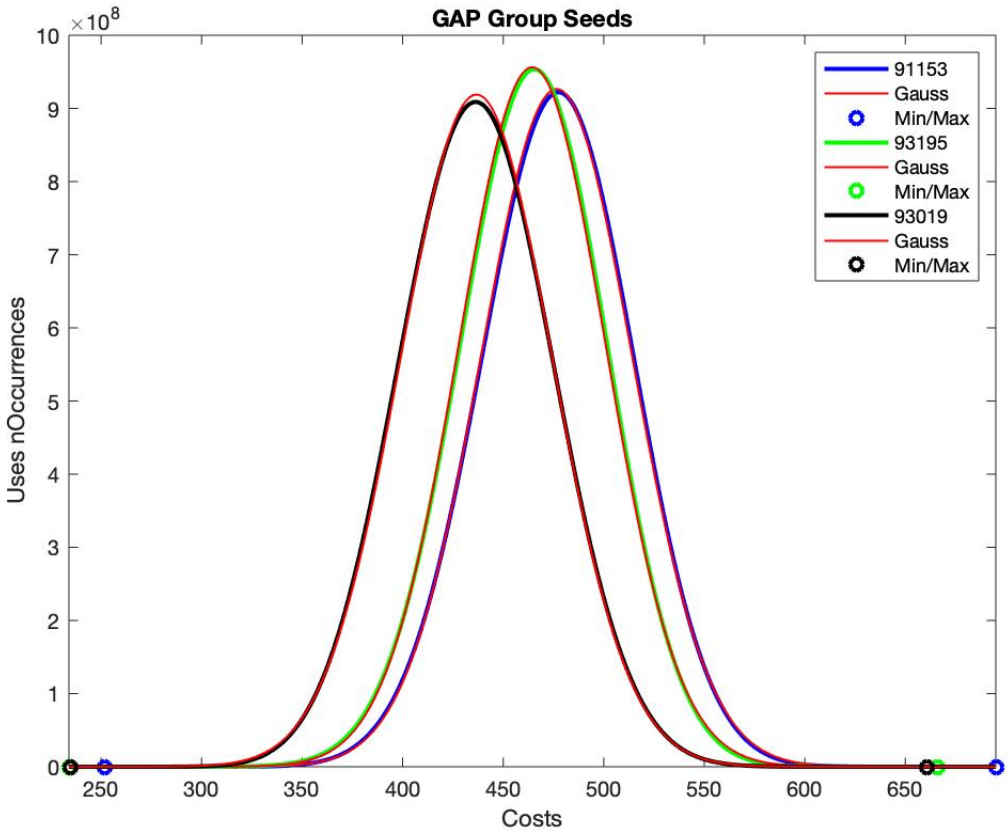

Figure 5: Occurrences according to the permutations costs, for n=14, using the Generate All Permutations method with the seeds correspondent to the group students numbers and respectively Gaussian

To compare the seeds it was decided to combine all the Costs/Occurrences results in a single graph. It's noticeable that even with different seeds, the means, minimums and maximums tend to be close. Also, in terms of probability, it is harder and harder to find a new minimum or maximum cost as it is diverged from the mean. The results can be approximated to a Gaussian Curve as it is seen in the graphic.

4.2 Results from Branch and Bound

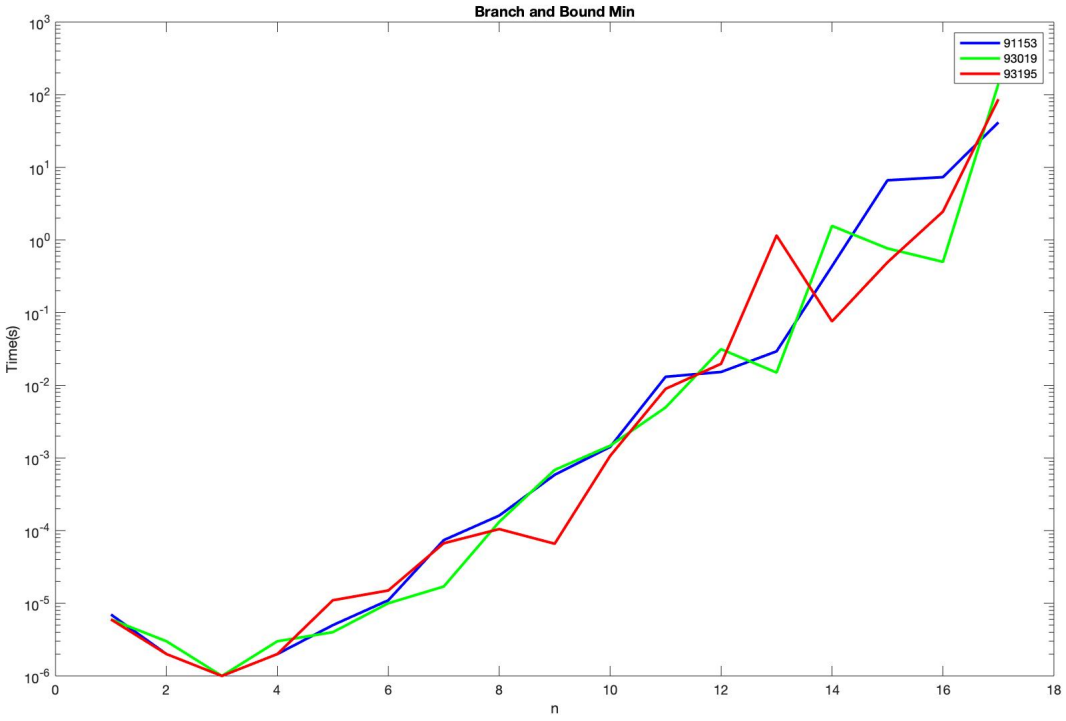

Figure 6: Execution time for n <= 17 using the Branch and Bound method for the minimum cost with the seeds correspondent to the group students numbers.

It is observed that this algorithm is pretty fast for n's until 16. For larger n's the execution time increases rapidly.

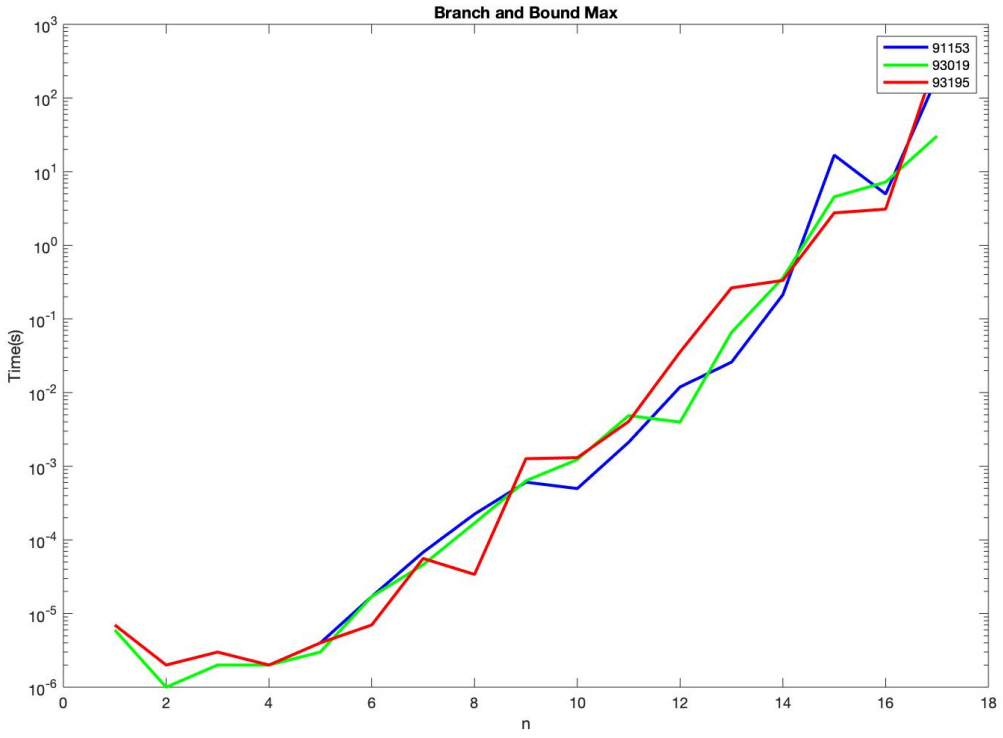

Figure 7: Execution time for n <= 17 using the Branch and Bound method for the maximum cost with the seeds correspondent to the group students numbers.

It is observed that it works similar to the Branch and Bound method for the minimum values, as expected. Fast for smaller n's and slower as n raises.

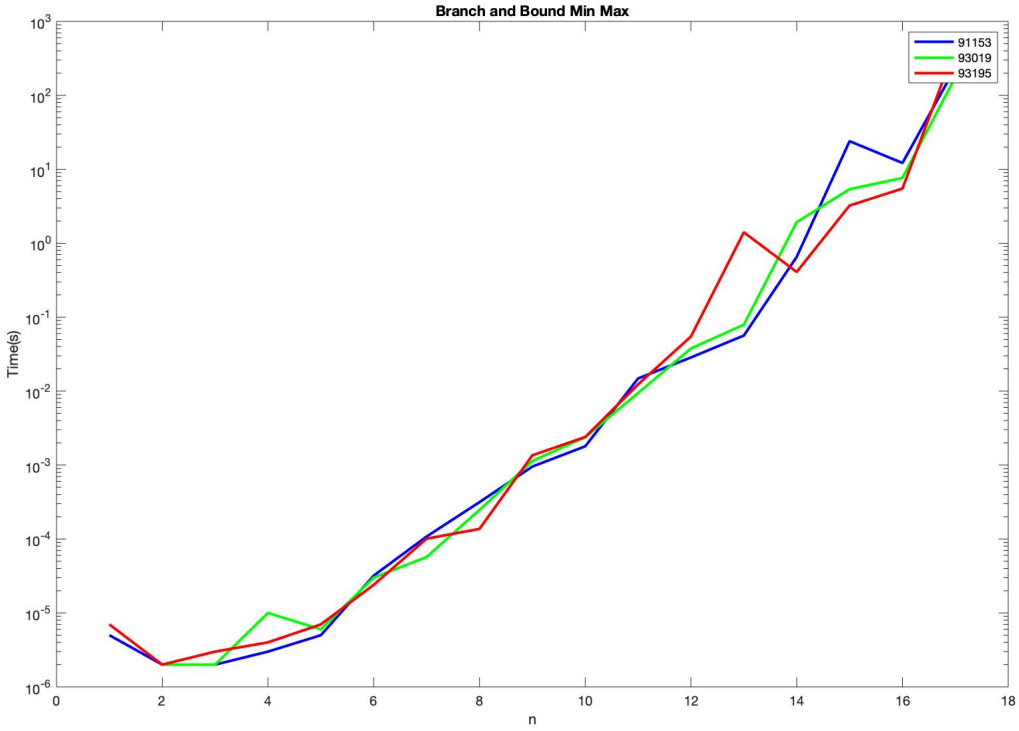

Figure 8: Execution time for n <= 17 using the Branch and Bound method for the minimum and maximum cost with the seeds correspondent to the group students numbers, at the same time.

The results are similar to the ones got for Branch and Bound minimum and Branch and Bound maximum.

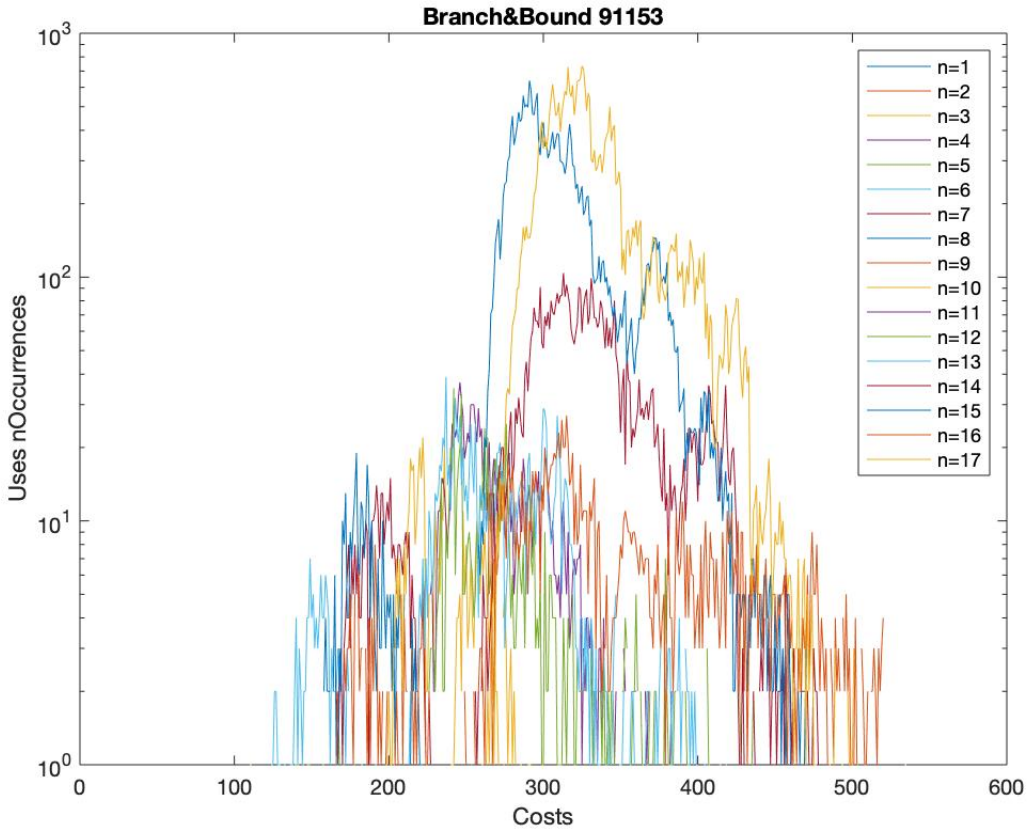

Figure 9: Occurrences according to the permutations costs, for each n, using the Branch and Bound method for the minimum cost with seed 91153

Observing the graphic, it can be concluded that the number of occurrences depends on how large is n number. It's noticeable that for a larger n, a big part of minimum costs are centered in the interval [200, 400]. It happened because at some point of the getting all permutation's process, it is known that the minimum cost saved is less than the sum with the next ideal minimum cost.

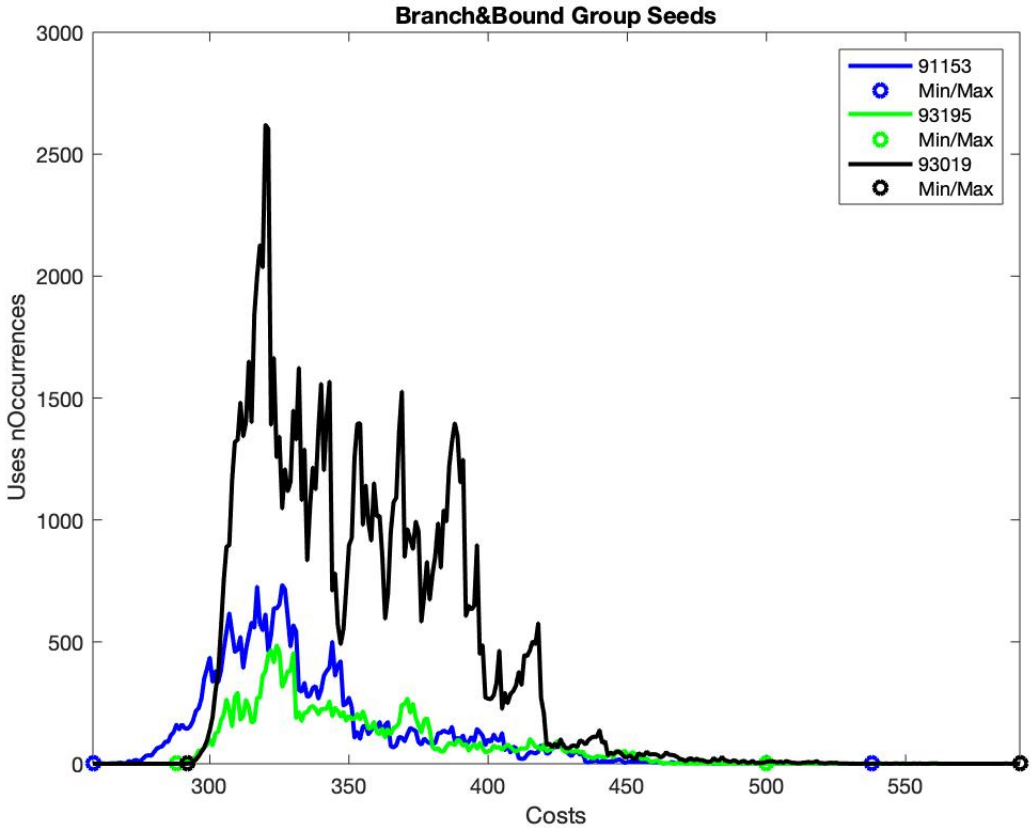

Figure 10: Occurrences according to the permutations costs, for n=17, using the Branch and Bound method for the minimum cost, with the seeds correspondent to the group students numbers

To compare the group students numbers seeds, the results were combined in a single graph with each seed Cost/Occurrences. It's clear that it varies a lot depending on which seed it's used. Note that the minimum costs are centered in the same range [300, 400], even with a different number of occurrences.

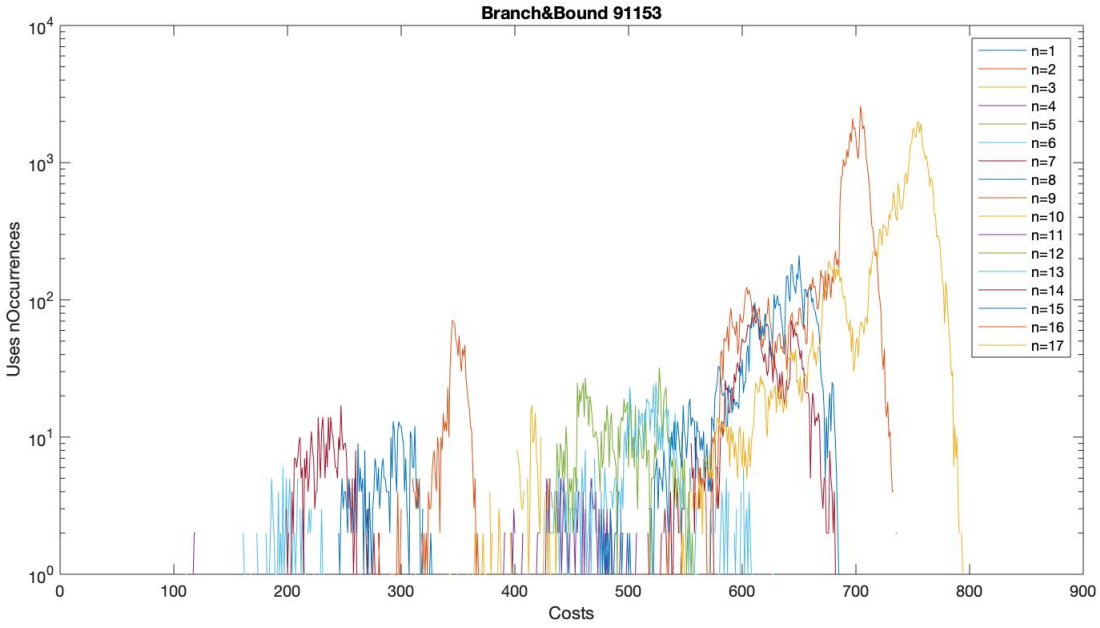

Figure 11: Occurrences according to the permutations costs, for each n, using the Branch and Bound method for the maximum cost with seed 91153

As it is raised the n, the results tend to have new maximum costs and a larger number of occurrences. Increasing the number n, means a significant growth of maximum costs.

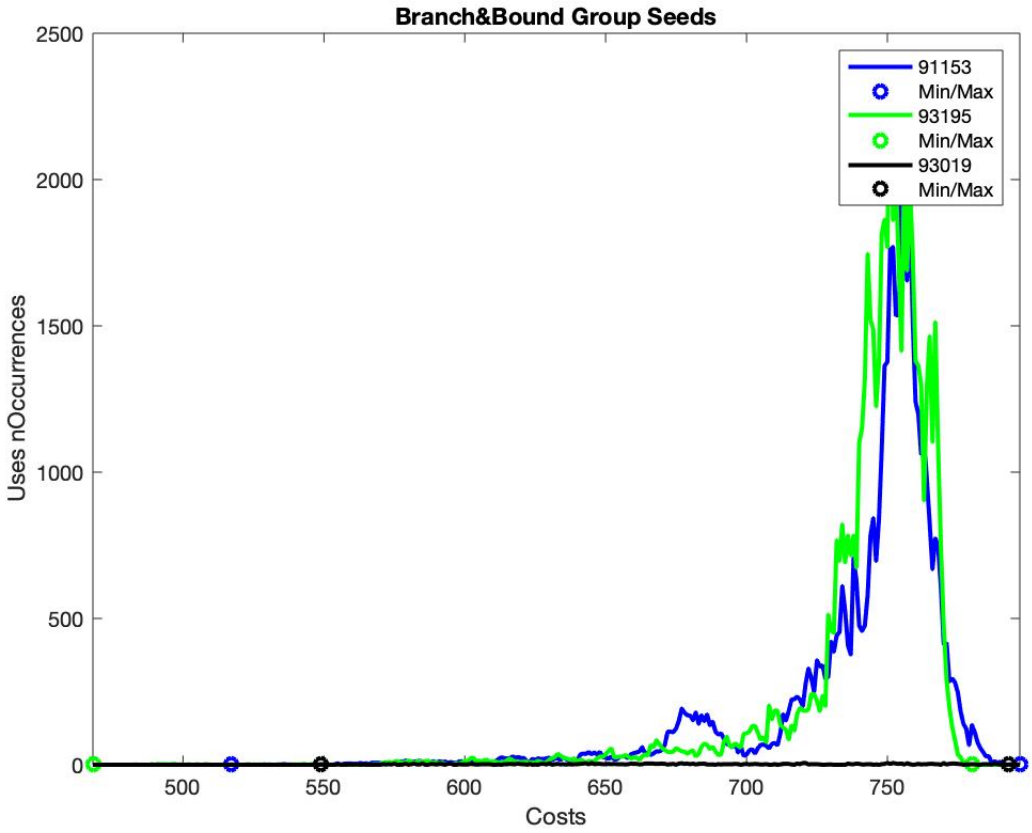

Figure 12: Occurrences according to the permutations costs, for n=17, using the Branch and Bound method for the maximum cost, with the seeds correspondent to the group students numbers

As it was done to the Branch and Bound method for the minimum cost, it was gathered all group students numbers seeds and the results were combined in a single graph with each seed Costs/Occurrences. One of the seeds has a low number of occurrences, but it can clearly be seen that the most frequent costs are gathered around [700, 800].

4.3 Comparing Generate All Permutations with Branch and Bound

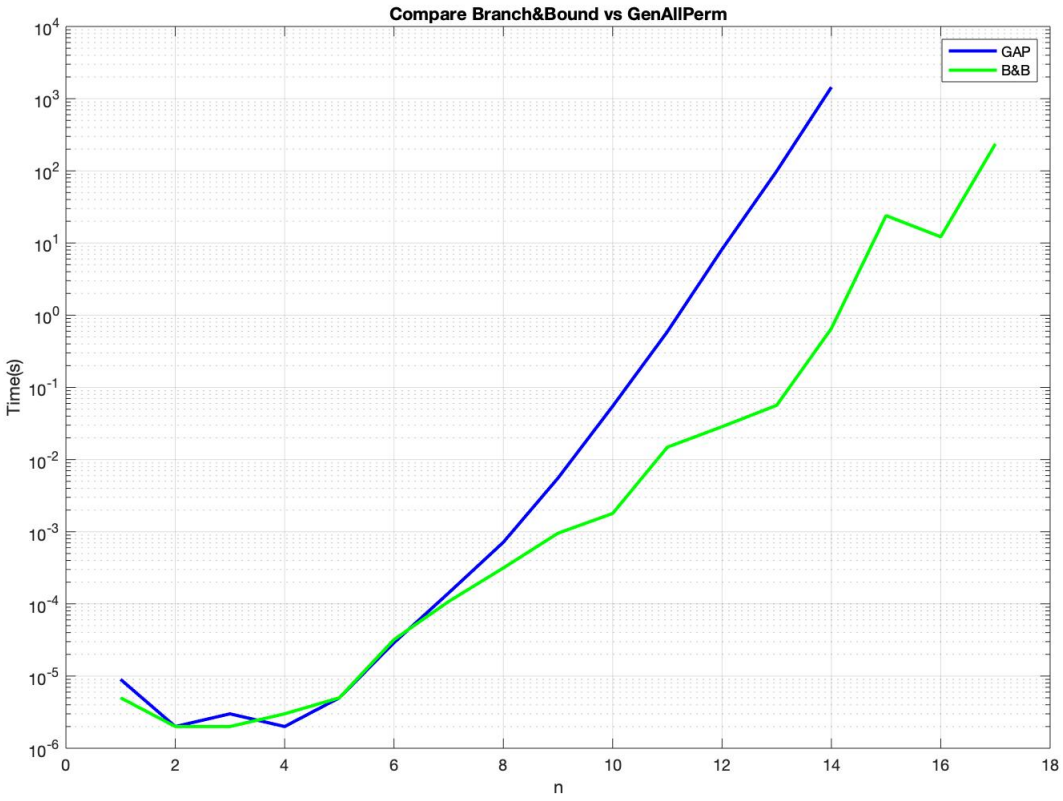

Figure 13: Comparing both methods: Generate All Permutations and Branch and Bound

Using small n values, the program time execution for both methods Generate All Permutations and Branch and Bounded are very similar. However, there is a big contrast on time performance between these methods when those n values are increased. For the n values represented, time performance difference is very low, but it can be predicted that for large n values, it can be more significant.
In this case, it can be concluded that Branch and Bound method can be a better choice for large scale computing.

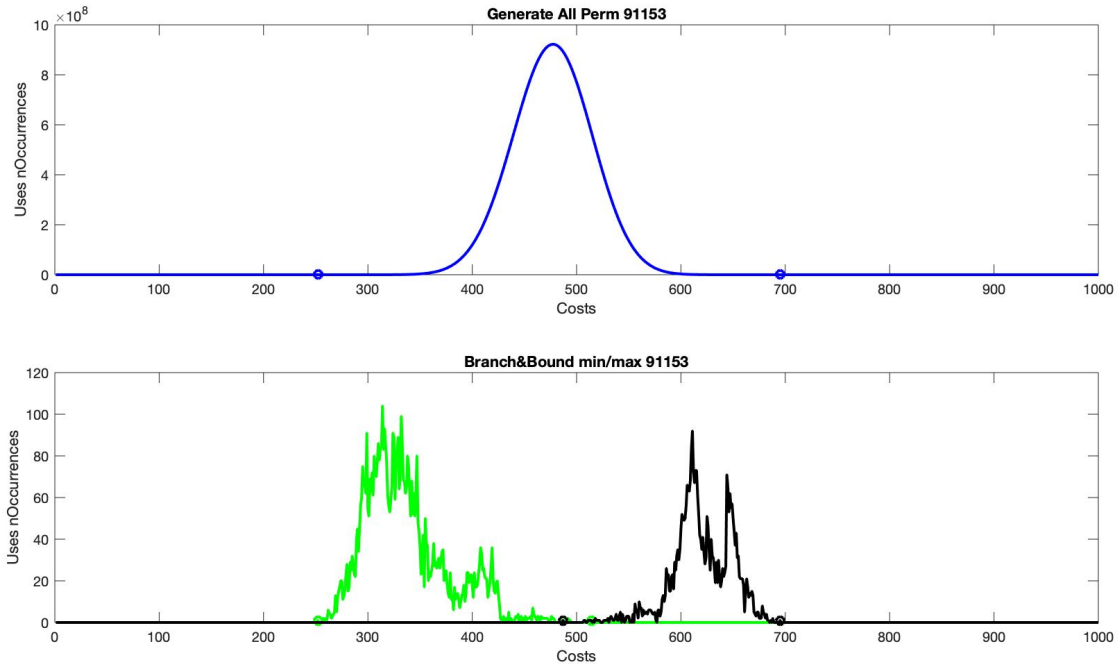

Figure 14: Comparison between the occurrences according to the permutations costs, for n=14, using the Branch and Bound method for the minimum and maximum cost and the Generate All Permutations method with the seed 91153

Considering the graphic, it is observed that the Generate All Permutations method is much closer to a normal distribution than the Branch and Bound method. The Branch and Bound method has more diverged values for the minimum and the maximum costs, while the Generate All Permutations method has more centered values.

4.4 Results from Random Permutations

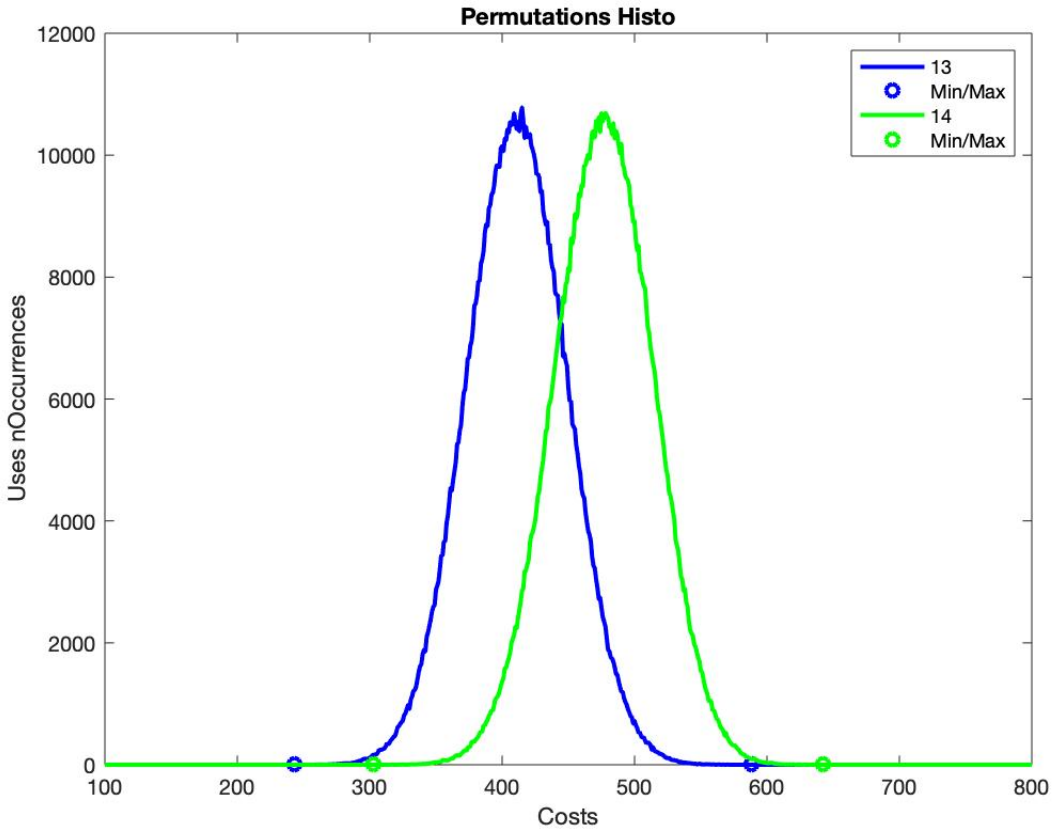

Figure 15: Occurrences according to the permutations costs, for n=13 and n=14, using the Random Permutations method for the minimum and maximum cost with the seed 91153 and running the function one million times

Considering the graphic, it is observed that the Random Permutations method is closer to a normal distribution for the permutations generated and this seed. Besides changing the n values, the results stayed close to a normal distribution. Also, it is possible to see that for n=13, the most frequent costs are gathered around [300, 500], and for n=14 around [400, 600].

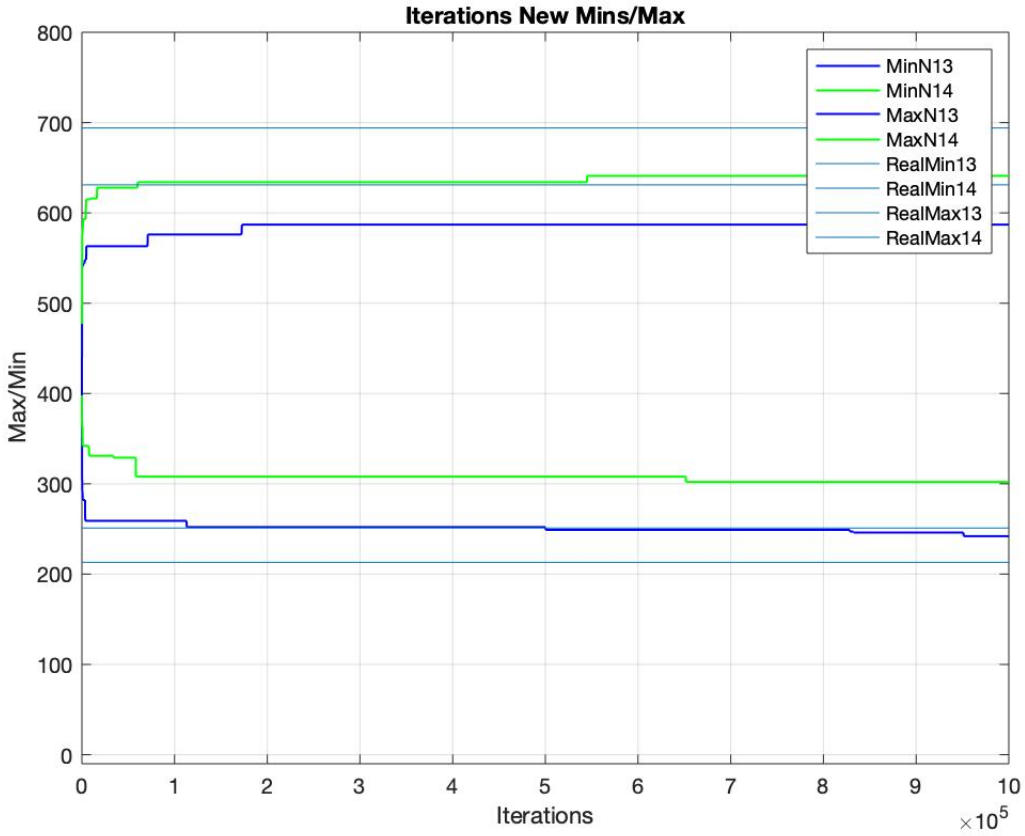

Figure 16: Maximum and minimum costs vs real maximum and minimum costs, for n=13 and n=14, using the Random Permutations method with the seed 91153 and running the function one million times

It is possible to compare the maximum and minimum cost resulted from this method (MinN13, MaxN13...) and the real ones (RealMin13, RealMax13...), for a specific n.
Using the information from the other methods and considering the graphic, it is observed that for the permutations generated running this function one million times, none of them was the permutation with the actual minimum or maximum cost for that n. Considering that, there were 13! possible permutations for n=13, and 14! possible permutations for n=14 but as it only ran one million times, the function, in the best scenario, passed by 1 million different permutations. Despite being a very large number, for this possible permutations it wasn't enough to go through the permutation that had the true maximum and minimum cost for that n.

5 Conclusion

This report introduced a set of approaches to the assignment problem. Not only it was learnt to look at a problem in several different ways but, by studying each solution, it was managed to assess what were the differences, the pros and the cons.

There were studied and learned various approaches to solve this problem: Generate All Permutations, Branch and Bound, Random Permutations and Greedy.

Using the Generate All Permutations method, it was concluded that, despite having accurate results, this method is time-consuming as the algorithm runs across all the possible cases.

The Branch and Bound method is much faster than the previous one (it ran instantly for smaller n's) because it can fix the values of the "current" maximum or minimum and not run across all the permutations with bigger or smaller costs. This way, some possible cases were bypassed.

Also, it was concluded that Random Permutations method is much faster than the Generate All Permutations when it ran less times than the total number of possible permutations. However, this brings a problem because it won't pass by all possible permutations and even the one that pass can be repeated, decreasing immensely the accuracy. If it runs more times than the total number of possible permutations, the method is just wasting computational power.

Finally, the Greedy method has a better performance than the others but uses a lazy criteria. It runs each line once, find the minimum value and remove the line and column where that minimum was located. By doing so, it is removed the size of the search for the next iteration but also, it lose precision on the final result.

In sum, every algorithm has its own value, some are slower but accurate and others are faster but not so accurate.

6 Bibliography

[1] SILVA, Tomás Oliveira e. **Lecture notes**: Algorithms and Data Structures (AED — Algoritmos e Estruturas de Dados) LEI, MIECT, 2019/2020.

[2] `https://cs.stackexchange.com/questions/72593/is-there-a-greedy-algorithm-to-solve-the-assignment-problem`

[3] `https://www.mathworks.com/matlabcentral/answers/111952-legend-button-displays-data1-data2-etc`