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

Elaborated on the curricular plan of the course Algoritmos e Estruturas de Dados, 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 ..................... for n=3 and n=5 using Brute Force method;

-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 is 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`';

```
1 int costAssignment(int n, int a[n]){
2   int permutation_cost = 0;
3
4   for (int i = 0; i < n; i++){
5     permutation_cost += cost[i][a[i]];
6   }
7   histogram[permutation_cost]++;
8   return permutation_cost;
9 }
```

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 in The Branch-and-Bound Method.

Let:

- (static int) '`cost[a][t]`' be the number of agents/tasks;
- (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 both the minimum and maximum cost of all the permutations, designated by the number of agents/tasks, in addition to 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's swapped 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 as well as the confirmation if it is the potential minimum or maximum cost, storing the actual permutation, if confirmed.

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

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;

```
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, if 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];
    }
  }
```

[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'.

```
24    if (permutation_cost < min_cost)
25    {
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 larger 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, if 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', 'max_int_cost' be the variables that hold the minimum and maximum cost of all costs;
- (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.

Version: Minimum Cost

```
1 static void generate_all_permutations_branch_and_bound_min(int n, int m,
2 int a[n], int partial_cost)
3 {
4   if (min_cost < (min_init_cost * (n - m) + partial_cost))
5   {
6     return;
7   }
8   else
9   {
```

```
    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, if 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) '`min_int_cost`', '`max_int_cost`' be the variables that hold the minimum and maximum cost of all costs;
- (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.

Version: Maximum Cost

```
1  static void generate_all_permutations_branch_and_bound_max(int n, int m,
2  int a[n], int partial_cost)
3  {
4    if (max_cost > (max_init_cost * (n - m) + partial_cost))
5      return;
6    else
7    {
8      if (m < n - 1)
9      {
10       for (int i = m; i < n; i++)
11       {
```

```
#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 won't be used again for the rest of the method. The estimated minimum cost of the matrix $n_{\times}n$ 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 column index will be the permutation array index and the line index will be the value of that permutation index. Let:

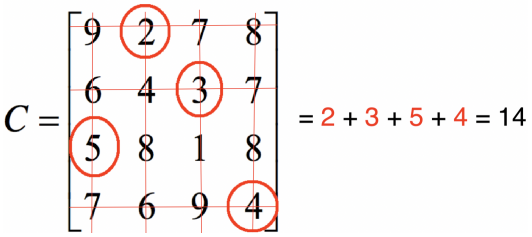

Figure 1: Example using a randomly-generated matrix C

- (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;

Version: Minimum Cost

```
1  static void greedy_method_min(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
7    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      final_min_cost += tmp_min_cost;
25    }
26    min_cost = final_min_cost;
27  }
28
```

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 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 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 column index will be the permutation array index and the line index will be the value of that permutation index.

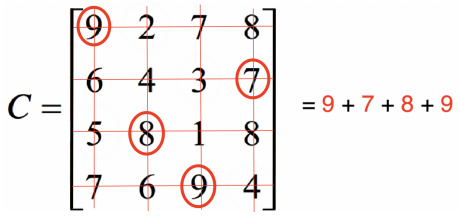

Figure 2: Example using a randomly-generated matrix C

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;

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;
11    for (int c = 0; c < n; c++) // column
12    {
13      if (cost[l][c] >= tmp_max_cost && binary_array[c] == 0)
```

```
14          {
15            tmp_max_cost = cost[l][c];
16            c_pos = c;
17          }
18        }
19        binary_array[c_pos] = 1;
20        final_max_cost += tmp_max_cost;
21      }
22      max_cost = final_max_cost;
23  }
24
```

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];
```

```
13
14 c = costs';
15 h = n_costs;
16 m = sum(c.*h)/sum(h);
17 v = sum((c-m).^2.*h)/sum(h);
18 p = 1/sum(h);
19 d = sqrt(2)*erfinv(2*p);
20
21 first1 = find(h, 1, "first");
22 last1 = find(h, 1, 'last');
23
24 plot_1 = plot(c, h, "b", "LineWidth", 2)
25 hold on
26 plot_2 = plot(c, exp(-(c-m).^2/(2*v))./sqrt(2*pi*v).*sum(h), "r", "LineWidth", 1)
27 hold on
28 plot_3 = plot([c(first1) c(last1)], [h(first1) h(last1)], "bo", "LineWidth", 2)
29 hold on
30
31 title('GAP Group Seeds')
32 xlabel('Costs')
33 ylabel('Uses nOccurrences')
34 xlim([first1 last1])
35
36 n_costs = T14_93195;
37 [lin, col] = size(n_costs);
38 costs = [1 : 1 : lin];
39
40 c = costs';
41 h = n_costs;
42 m = sum(c.*h)/sum(h);
43 v = sum((c-m).^2.*h)/sum(h);
44 p = 1/sum(h);
45 d = sqrt(2)*erfinv(2*p);
46
47 first2 = find(h, 1, "first")
48 last2 = find(h, 1, 'last')
49
50 if first2 < first1
51     first = first2
52 else
53     first = first1
54 end
55
56 if last2 > last1
57     last = last2
58 else
59     last = last1
60 end
61
62 plot_4 = plot(c, h, "g", "LineWidth", 2)
63 hold on
64 plot_5 = plot(c, exp(-(c-m).^2/(2*v))./sqrt(2*pi*v).*sum(h), "r", "LineWidth", 1)
65 hold on
66 plot_6 = plot([c(first2) c(last2)], [h(first2) h(last2)], "go", "LineWidth", 2)
67 hold on
68
69 title('GAP Group Seeds')
70 xlabel('Costs')
71 ylabel('Uses nOccurrences')
72 xlim([first last])
73
74 n_costs = T14_93019;
```

```
[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);

first3 = find(h, 1, "first")
last3 = find(h, 1, 'last')

if first3 < first
    first = first3
end

if last3 > last
    last = last3
end

plot_7 = plot(c, h, "black", "LineWidth", 2)
hold on
plot_8 = plot(c, exp(-(c-m).^2/(2*v))./sqrt(2*pi*v).*sum(h), "r", "LineWidth", 1)
hold on
plot_9 = plot([c(first3) c(last3)], [h(first3) h(last3)], "blacko", "LineWidth", 2)
hold on

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")

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

```

Therefore, it was created the following:

4.1 Occurrences according to the costs

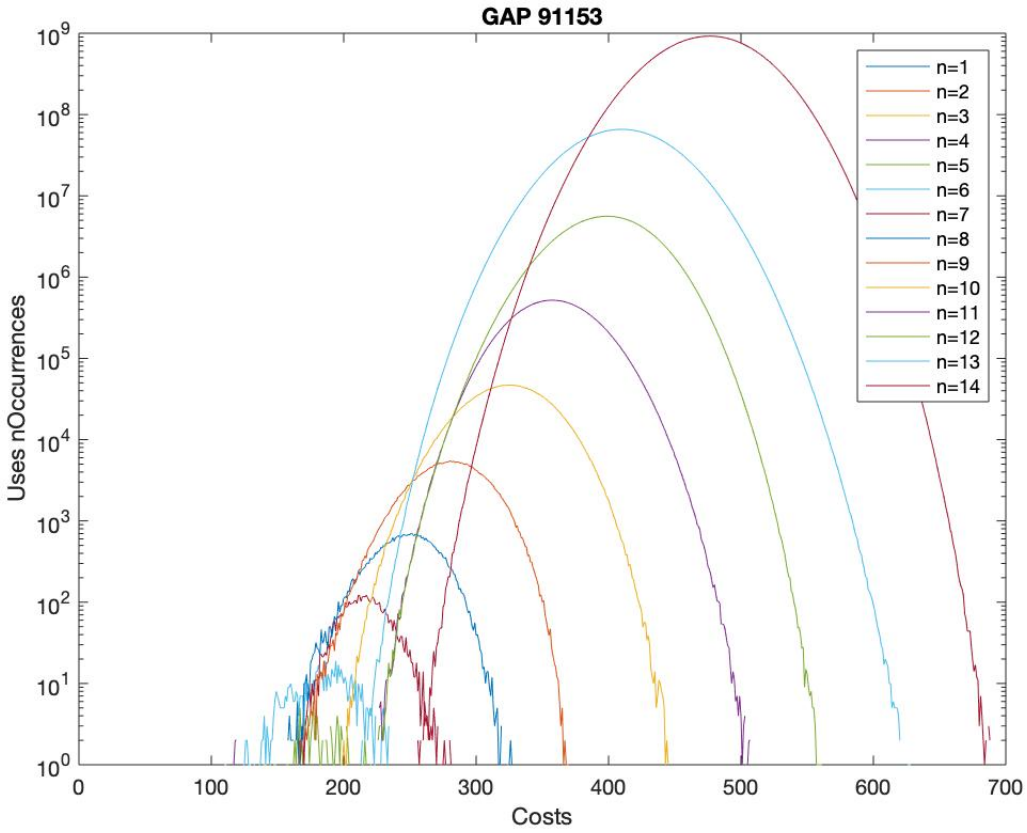

Figure 3: Occurrences according to the costs, for each n, using the Method Generate All Permutations with seed 931153

(Explicar por alto o gráfico) - É possível verificar que para n's maiores atingimos custos maiores. O número de ocorrências é maior, quanto maior for o n.

4.2 Time spent according to the n agents/tasks

5 Conclusion

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`