ASSIGNMENT PROBLEM

Engenharia Informática

Leandro Silva 93446 / Mário Silva 93340

2019/2020

Contribution percentage: 50/50

Introduction

The problem instance **n** agents and **n** tasks. Any agent **a** can be assigned to perform any task **t**, incurring the cost $\boldsymbol{C_{a,t}}$ that may vary depending on the agent-task assignment. It is required to assign at most one agent to each task and at most one task to each agent, in such a way that the total cost of the assignment is minimized.

Given two sets, *A* and *T*, of equal size, together with a weight function $C : A \times T \rightarrow \mathbf{R}$. Find a bijection $f : A \rightarrow T$ such that the cost function:

$$\sum_{a \in A} C(a, t(a))$$

is minimized.

Usually the weight function is viewed as a square real-valued matrix *C*, so that the cost function is written down as:

$$\sum_{a \in A} C_{a,t(a)}$$

In the C programming language, the matrix *C* will be represented as a two dimensional array, **cost[][]**, where the first indices will be the agent index and the second the task index. The values for this matrix will be randomly chosen between 3 and 60.

Solutions and Used Methods

Brute Force General Permutations

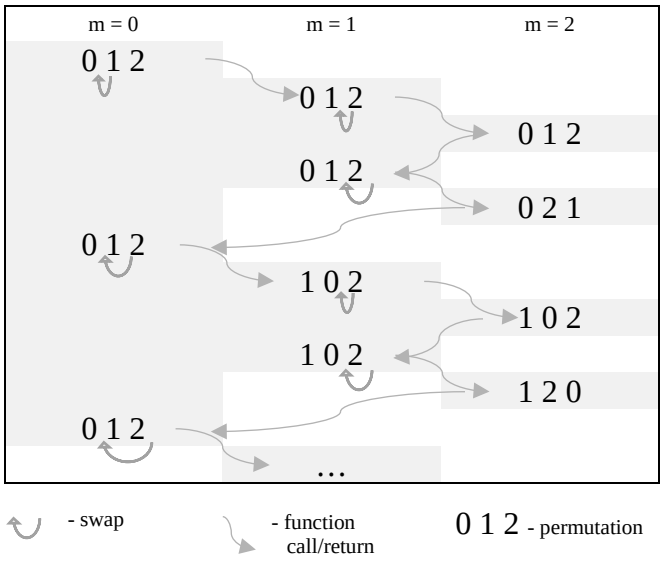
This method consists of systematically enumerating all possible permutation for the assignment solution and checking which permutation satisfies the problem's statement – finding the assignments whose cost is the lowest and the highest.

While a brute-force search is simple to implement, and will always find a solution if it exists, its time execution tends to grow very quickly as the size of the problem increases (combinatorial explosion).

To enumerate all possible permutation, the code that was given will start by an initial permutation that is created outside of the function:

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

This permutation is passed as an argument to the function, **generate_all_permutations()**, where it will have its indices swaped into new permutations recursively.



This scheme shows an example of how function reorganizes the initial permutation, 0 1 2, into the **n!** possible permutations. When **m** = 0 or 1, the function does a 'for' cycle to swap every **m** to **n** elements of the permutation with the **m** element and call the next recursion (lines 3 to 13).

For **m** = 2, the reorganization is completed. Thus, it will sum all of the costs from the permutation into **total_cost** and check whether or not it is better that the current solution. If so, the permutation will be stored (in **min_cost_assignment**, for the minimum solution) and the

global variable for the solution (**min_cost**, for the minimum solution), will be equal to **total_cost** (lines 14 to 34).

The complexity of this algorithm is O(n*N!) because there's a for cycle that goes up to N and then inside of this loop, the function is called again with recursion.

```
1   static void generate_all_permutations(int n, int m, int a[])
2   {
3       if(m < n - 1)
4       {
5           for(int i = m;i < n;i++)
6           {
7   #define swap(i,j)  do { int t = a[i]; a[i] = a[j]; a[j] = t; } while(0)
8               swap(i,m);
9               generate_all_permutations (n,m + 1,a);
10              swap(i,m);
11  #undef swap
12          }
13      }
14      else
15      {
16          n_visited++;
17
18          int total_cost = 0;
19          for( int i=0; i<n;i++){
20              total_cost += cost[i][a[i]];
21          }
22          if (total_cost<min_cost){
23              min_cost=total_cost;
24              for(int j = 0; j < n; j++)
25                  min_cost_assignment[j] = a[j];
26          }
27          if (total_cost>max_cost){
28              max_cost=total_cost;
29              for(int j = 0; j < n; j++)
30                  max_cost_assignment[j] = a[j];
31          }
32
33          histogram[total_cost] += 1;
34      }
35  }
```

Brute Force Random Permutations

The Random Permutations algorithm basically does a bunch of permutations and then calculates the costs, we had already a function given by the professor that did a random permutation and then we calculated the costs just like the brute force general permutation. It was asked to do about 1 000 000 random permutations so we just did a cycle from 0 to 1 000 000, given that an N has only N! permutations, the chance of getting the right minimum cost or maximum decreases a lot when N>=10, it doesn't mean that for N=9 it will get the right cost, but the chances are higher than N>=10 because 10! is already greater than 1 000 000 so it will never do all the possible permutations like the brute force general permutation does, therefore it might not get the right permutation for the minimum and maximum cost.

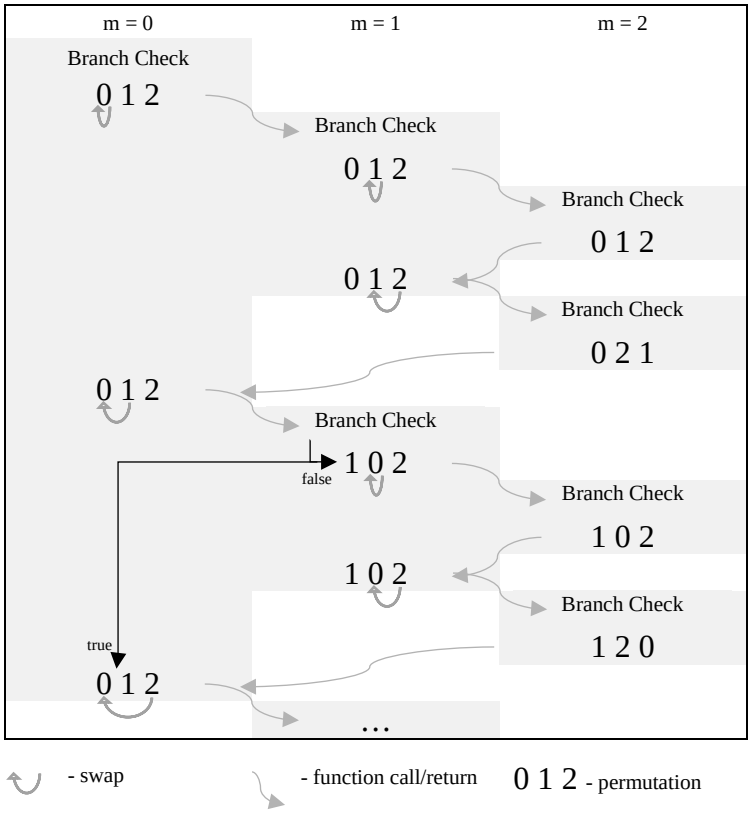
The complexity of this algorithm is O(N) because in the function **random_permutations()** there is a cycle that goes up to N.

Branch-and-Bound

Branch-and-bound is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization.

The algorithm explores *branches* of a rooted tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search, as in Brute Force General Permutations.



The Branch-and-Bound used in our code is essentially the General Permutations with our branch check.

This branch check is a piece of code which will decide to continue or discard the initial part of a permutation, according to the actual lower bound, the **min_cost**.

Here's an example:

If our current **min_cost** is from the permutation t(0) = 0, t(1) = 2, t(2) = 1, with cost 3 + 5 + 7 = 15, then, if we had next a permutation started by t(0) = 1, with cost 12, it would matter study the rest, as the best minimum that it could give is 12 + 3 + 3 = 18, which is greater than 15. Notice that 3 is the minimum value cost.

Thus, the code should be something similar to this:

```
1   static void generate_all_permutations_branch_and_bound(int n, int m, int a[], int partial_cost)
2   {
3       if(min_cost < partial_cost + 3*(n – m))
4           return;
5
6       if(m < n - 1)
7       {
8           for(int i = m;i < n;i++)
9           {
10  #define swap(i,j)  do { int t = a[i]; a[i] = a[j]; a[j] = t; } while(0)
11              swap(i,m);
12              generate_all_permutations_branch_and_bound(n,m + 1,a, partial_cost + cost[m][a[m]]);
13              swap(i,m);
14  #undef swap
15          }
16      }
17      else
18      {
19          int total_cost = partial_cost + cost[m][a[m]];
20          if(min_cost > total_cost){
21              min_cost = total_cost;
22              for(int j = 0; j < n; j++)
23                  min_cost_assignment[j] = a[j];
24          }
25
26          n_visited++;
27      }
28  }
```

The differences from the **generate_all_permutations()** are the function 4th argument, **partial_cost**, and the first 'if' (line 3), which represents the branch check. The **partial_cost** is the sum of all cost values from part of the permutation. It is only used to calculate the best minimum of the branch, by adding it the number of the remaining costs (**n – m**) to complete a solution times 3:

```
partial_cost + 3*(n -m)
```

If the best minimum of a branch is greater than the current **min_cost**, it's discarded by prematurely returning the function.

When **m =** 0, the branch check is pointless, as the branch root will never be discarded.

If we wanted the maximum cost, it would only be necessary to modify some lines of code. The global variable for the upper *bound* would be **max_cost** and the number of the remaining costs would be multiplying by 60 (maximum cost value). The 3rd line should be something like:

```
if(max_cost > partial_cost + 60*(n – m))
```

Branch-and-Bound (improved)

The Branch-and-Bound, as it is, is significantly faster than the General Permutations. However, it can be much faster.

To discard branches, we are calculating its best solution by assuming that its remaining costs are equal to 3. Instead, what if we assumed that its remaining costs were equal to the matrix's minimum cost? It would discard even more branches and surely improve the fastness. This idea grows into the following thought: what if we stored the minimum cost of each matrix's row in an array, and used it to calculate the branch's best solution more precisely? And this was what we ended making.

By creating a global array, **min_cost_row[]**, and calling the function above, we are able to store the minimum cost from each 'row' of **cost[][]** in **min_cost_row[]**.

```
1   static void costs_row(int n)
2   {
3       int min;
4       for(int a = 0;a < n; a++){
5           min = plus_inf;
6           for(int t = 0; t < n; t++){
7               if(min > cost[a][t])
8                   min = cost[a][t];
9           }
10          min_cost_row[a] = min;
11      }
12  }
```

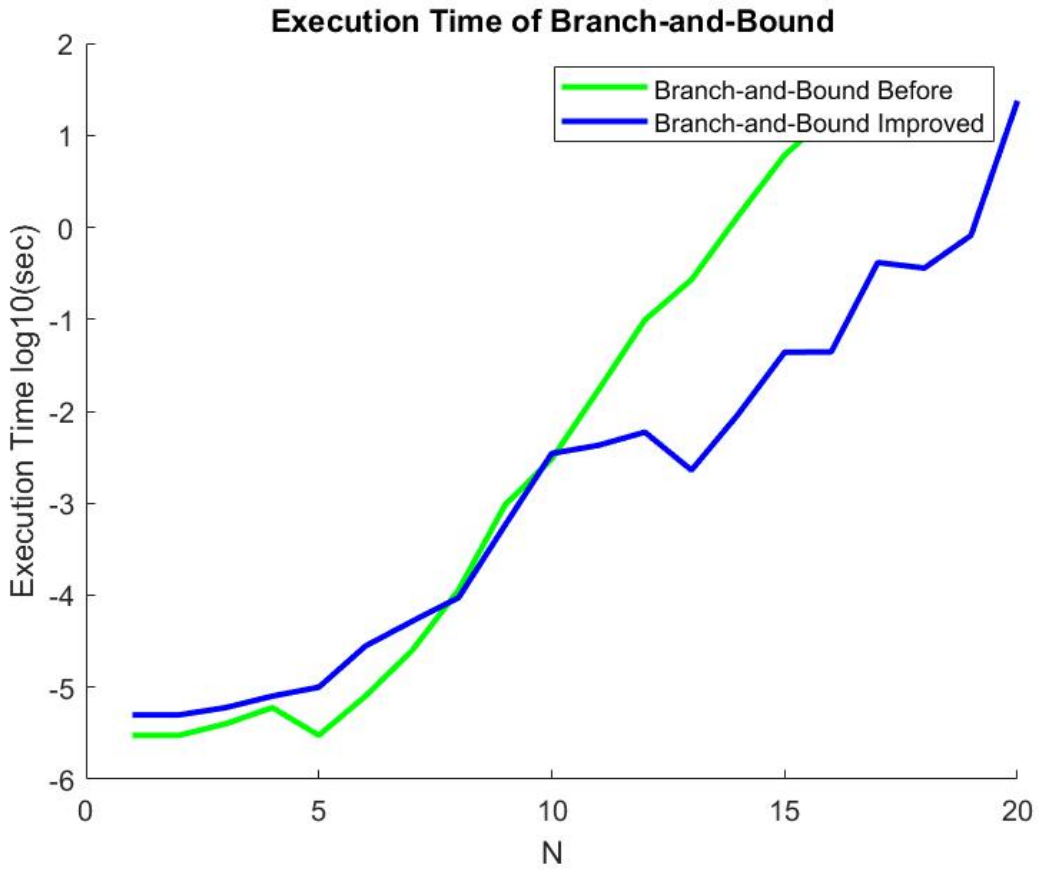
Next, inside the function **min_branch_and_bound()**, we sum all the **n – m** remaining costs according to the minimums by row. This can be done by using a 'for' cycle that sums the costs stored in **min_cost_row[]** since the index **m** till **n** (line 16), and then we add it to the **partial_cost** to compare with the current **min_cost** (line 18).

```
13  static void min_branch_and_bound(int n, int m, int a[], int partial_cost)
14  {
15      int min = 0;
16      for(int i = m; i < n; i++){ min += min_cost_row[i]; }
17
18      if(min_cost < partial_cost + min)
19          return;
    …
```

To make things easier, we created another function to call the last two in the right order and we took the opportunity to add a call to **max_branch_and_bound()**, a similar function of **min_branch_and_bound()** that instead calculates the maximum solution.

```
20      static void generate_all_permutations_branch_and_bound(int n, int m,int a[n], int partial_cost)
21      {
22              costs_row(n);
23              min_branch_and_bound(n, m, a, partial_cost);
24              max_branch_and_bound(n, m, a, partial_cost);
25      }
```

From the graphic bellow we can see how much faster is the Improved Branch-and-Bound (note that the Branch-and-Bound Before only calculated the minimum as the Improved calculates both maximum and minimum).



Hungarian Method

Besides the last methods, we tried to implement one of the bests methods known to resolve the assignment problem – the Hungarian Method. The reason why we chose this method was its impressive complexity of O(n3). This method has the following steps:

1. For each row, find the smallest element and subtract it from every element in its row.
2. Do the same (as step 1) for all columns.
3. Cover all zeros in the matrix using minimum number of horizontal and vertical lines.
4. *Test for Optimality:* If the minimum number of covering lines is n, an optimal assignment is possible and we are finished. Else if lines are lesser than n, we haven't found the optimal assignment, and must proceed to step 5.
5. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to step 3.

The step 1 and 2 were easy to implement. Firstly, we had to create a global copy of **cost[][]**, **int cost2[max_n][max_n]**, to make our changes. Then, we just needed to use a 'for' cycle going through each row twice: one to find its smallest element (lines 8-10, 18-20) and the other to subtract all elements with the smallest found (lines 11-12, 21-22). When done with the rows, we had to do the same for the columns.

```
1   static void hungarian_algorithm(int n)
2   {
3           int min;
4
5           // subtracts minimum of each row
6           for(int i = 0; i < n; i++){
7                   min = plus_inf;
8                   for(int j = 0; j < n; j++){
9                           if(cost2[i][j] < min){ min = cost2[i][j]; }
10                  }
11                  for(int j = 0; j < n; j++)
12                          cost2[i][j] -= min;
13          }
14
15          // subtracts minimum of each column
16          for(int i = 0; i < n; i++){
17                  min = plus_inf;
18                  for(int j = 0; j < n; j++){
19                          if(cost2[j][i] < min){ min = cost2[j][i]; }
20                  }
21                  for(int j = 0; j < n; j++)
22                          cost2[j][i] -= min;
23          }
            …
```

Now comes the difficult part. How do I cross the zeros from **cost2[][]** with the minimum number of lines? The common sense tells that we need to start crossing a line where it will cover the maximum number of zeros: the fewer uncovered zeros remaining, the fewer lines remaining to cover them, right? No, not always. An example:

1	0	1	1	1	1	0	{2}
1	1	1	0	1	1	1	{1}
1	1	0	0	0	1	1	{3}
0	1	1	1	1	0	1	{2}
1	1	0	1	1	1	1	{1}
0	1	1	1	1	1	1	{1}
1	1	1	1	0	1	1	{1}
{2}	{1}	{2}	{2}	{2}	{1}	{1}	

1	0	1	1	1	1	0	{2}
1	1	1	0	1	1	1	{1}
x	x	x	x	x	x	x	{0}
0	1	1	1	1	0	1	{2}
1	1	0	1	1	1	1	{1}
0	1	1	1	1	1	1	{1}
1	1	1	1	0	1	1	{1}
{2}	{1}	{1}	{1}	{1}	{1}	{1}	

When repeatedly crossing the rows/columns that have the greatest number of zeros, till there's no more, will achieve this matrix:

x	x	x	x	x	x	x	{0}
x	x	x	x	x	x	x	{0}
x	x	x	x	x	x	x	{0}
x	x	x	x	x	x	x	{0}
x	x	x	x	x	x	x	{0}
x	x	x	x	x	x	x	{0}
x	x	x	x	x	x	x	{0}
{0}	{0}	{0}	{0}	{0}	{0}	{0}	

It required 7 lines to cover all zeros, the same as **n**, in this case. This would mean that it was ready to proceed to step 4. However, the real total number of lines needed are 6, as it follows:

x	x	x	x	x	x	x	{0}
1	1	x	x	x	1	1	{0}
1	1	x	x	x	1	1	{0}
x	x	x	x	x	x	x	{0}
1	1	x	x	x	1	1	{0}
x	x	x	x	x	x	x	{0}
1	1	x	x	x	1	1	{0}
{0}	{0}	{0}	{0}	{0}	{0}	{0}	

Building an algorithm with this thought would lead to mistakes in the program for some cases like this.

We know for sure that when a zero is alone in its row, it can be cover immediately with a vertical line, whether it has zeros in its column or not (doesn't make a difference). The same applies to zeros alone in its column being covered by horizontal lines. So, we came up with this:

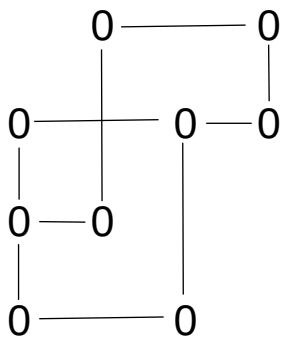
```
24  while(1){
25          // covers all zeros with the minimum number of lines
26          count3 = -1; count2 = 0; count = 0;
27          do{
28                  stop = true;
29                  for(int i = 0; i < n; i++){
30                          for(int j = 0; j < n; j++){
31                                  if(cost2[i][j] == 0 && lines[i][j] == 0){
32                                          stop = false;
33                                          zeros = 0b0000;     //  represent if the zero has (1) or not (0) zeros in its 'neighborhood'
34                                                              // the order is the following: up, down, left and right
35                                          for(int k = 0; k < n; k++){
36                                                  if(cost2[k][j] == 0 && lines[k][j] == 0){
37                                                          if(k < i)
38                                                                  zeros |= 0b1000;     // has upwards
39                                                          else if(k > i)
40                                                                  zeros |= 0b0100;     // has downwards
41                                                  }
42                                                  if(cost2[i][k] == 0 && lines[i][k] == 0){
43                                                          if(k < j)
44                                                                  zeros |= 0b0010;     // has leftwards
45                                                          else if(k > j)
46                                                                  zeros |= 0b0001;     // has rightwards
47                                                  }
48                                          }
49                                          if((zeros & 0b1100) == 0b0000){
50                                                  // covers a row
51                                                  for(int k = 0; k < n; k++)
52                                                          lines[i][k] = lines[i][k] == 1? 2 : -1;
52                                                  count++;
54                                                  }
55                                          else if((zeros & 0b0011) == 0b0000){
56                                                  // covers a column
57                                                  for(int k = 0; k < n; k++)
58                                                          lines[k][j] = lines[k][j] == -1? 2 : 1;
59                                                  count++;
60                                          }
61
62                                          else if(count3 == count2){       // if nothing has changed from last two while iterations
63                                                  // ? ? ?
64                                          }
65                                  }
66                          }
67                  }
68                  count3 = count2;
69                  count2 = count;
70          }while(!stop);

            …
```

We created another global 'matrix', **lines[][]**, to take record the lines we crossed so far. Before the 'while' in line 24, we filled **lines[][]** with zeros. When we want to cover a row, we put -1's in the row's cells; to cover a column, we put 1's. In the lines intersections we put

2 (lines 50-52, 56-58). To decide where to cross the lines, we created an inner function global variable, **int zeros**, that stores the information relatively to the existence of zeros upwards, downwards, leftwards, and rightwards for each zero (lines 33-48). It's more information than we need, as the existence of zeros in the row/column was enough. We did like that because we thought we were going to need this information later step 4. The principle is: if it has no zeros upwards and downwards, cover a row (line 49); if it has no zeros leftwards and rightwards, cover a column (line 55).

This will be done till there's no zeros in uncovered in **cost2[][]** from the moment it gets into the 'do while' (line 27), because the Boolean variable **stop** (which stops the loop) won't be true unless there's at least one cell from **cost2[][]** that verifies the 'if' in line 31.

Nevertheless, there will be cases where it won't be possible to cover any further lines. Cases where all zeros have a neighbour in the row and column, like the following scheme.



When this happens, **count3** (number of lines 2 iterations before) is equal to **count2** (number of lines 1 iteration before). Thus, the program will get inside the 'if' statement in line 62.

Here, we don't know what to do more. If we covered the maximum zeros possible, it will lead to the mistake we detailed above. But we did it anyways to see how far it would go before ending up in an endless loop (the furthest we achieved was till **n** = 97 for the seed 12, and it was instantaneous).

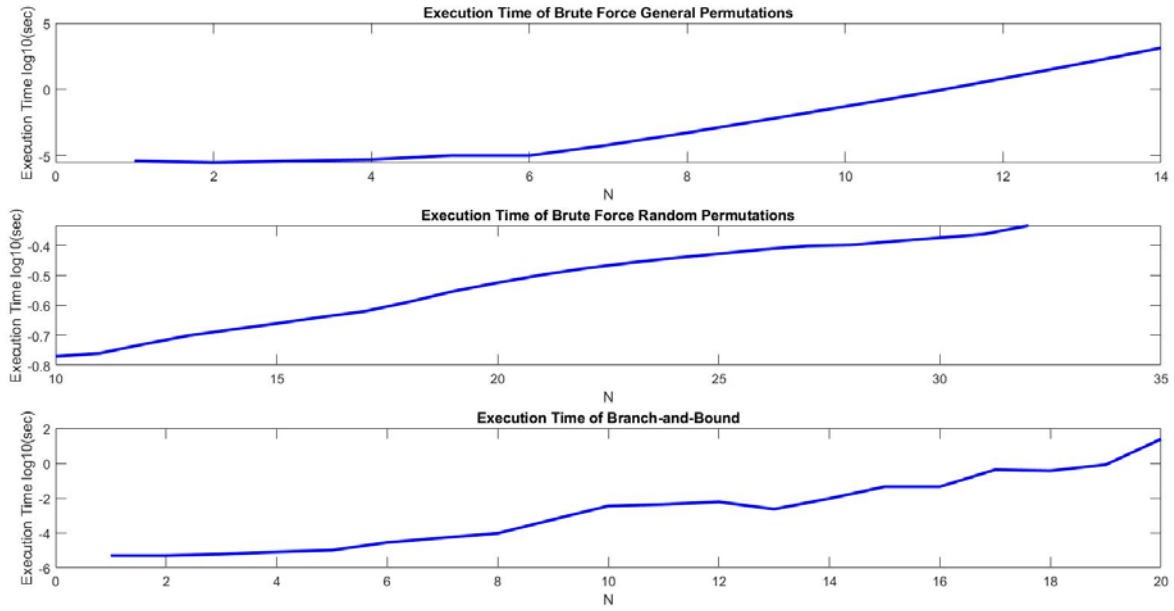
Step 5 has no problems at all. Regarding the Hungarian Method, it is done when the minimum number of lines is different from **n** (line 84-106). If it is equal, then it breaks the 'while' loop in line 24 and determines the optimal assignment.

```
81              if(count == n)          // here, count is the number of lines
82                      break;
83
84              else{
85                      // finds the smallest uncovered entry
86                      min = plus_inf;
87                      for(int i = 0; i < n; i++){
88                              for(int j = 0; j < n; j++){
89                                      if(lines[i][j] == 0 && cost2[i][j] < min){
90                                              min = cost2[i][j];
91                                      }
92                              }
93                      }
94                      for(int i = 0; i < n; i++){
95                              for(int j = 0; j < n; j++){
96                                      // subtracts the entry from all uncovered rows
97                                      if(lines[i][j] == 0 || lines[i][j] == 1)
98                                              cost2[i][j] -= min;
99                                      // adds the entry to all covered columns
100                                     if(lines[i][j] == 1 || lines[i][j] == 2)
101                                             cost2[i][j] += min;
102                                     // resets lines
103                                     lines[i][j] = 0;
104                             }
105                     }
106             }
107     }
```

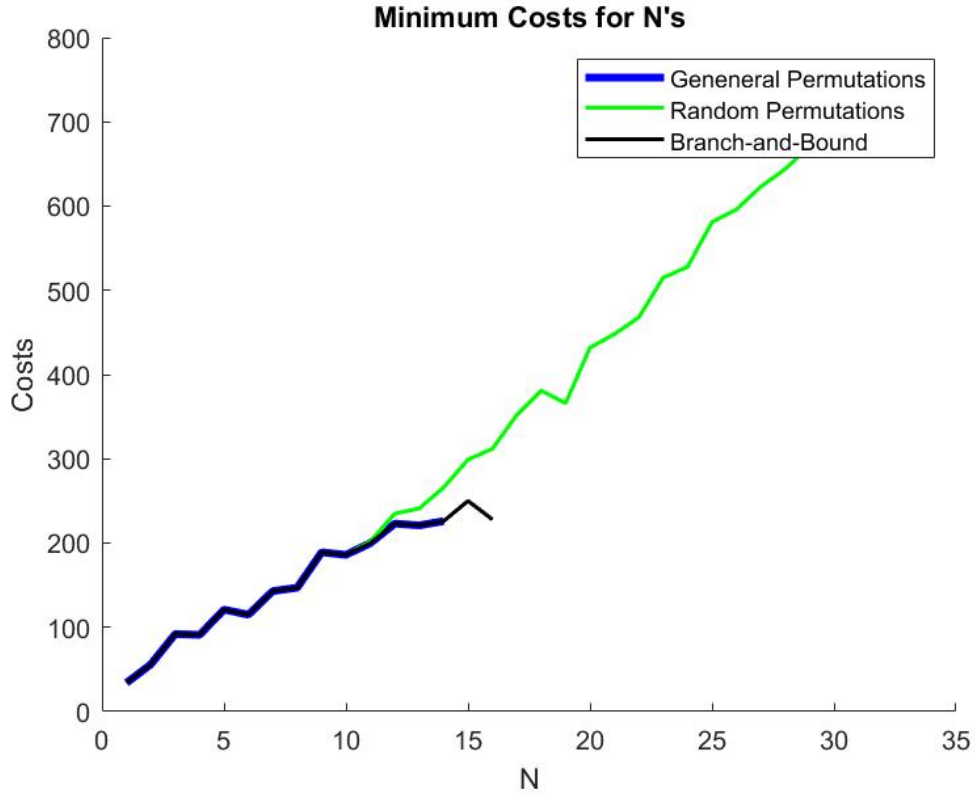
We're not going to detail the optimal assignment, as it is very similar to covering all zeros with the minimum number of lines. We made it with the same thought process and, subsequently, with the same mistakes. We've discovered very recently a possible way of doing this two steps - a bipartite perfect matching – but we have no certain that it will work and we had no time to do further researches.

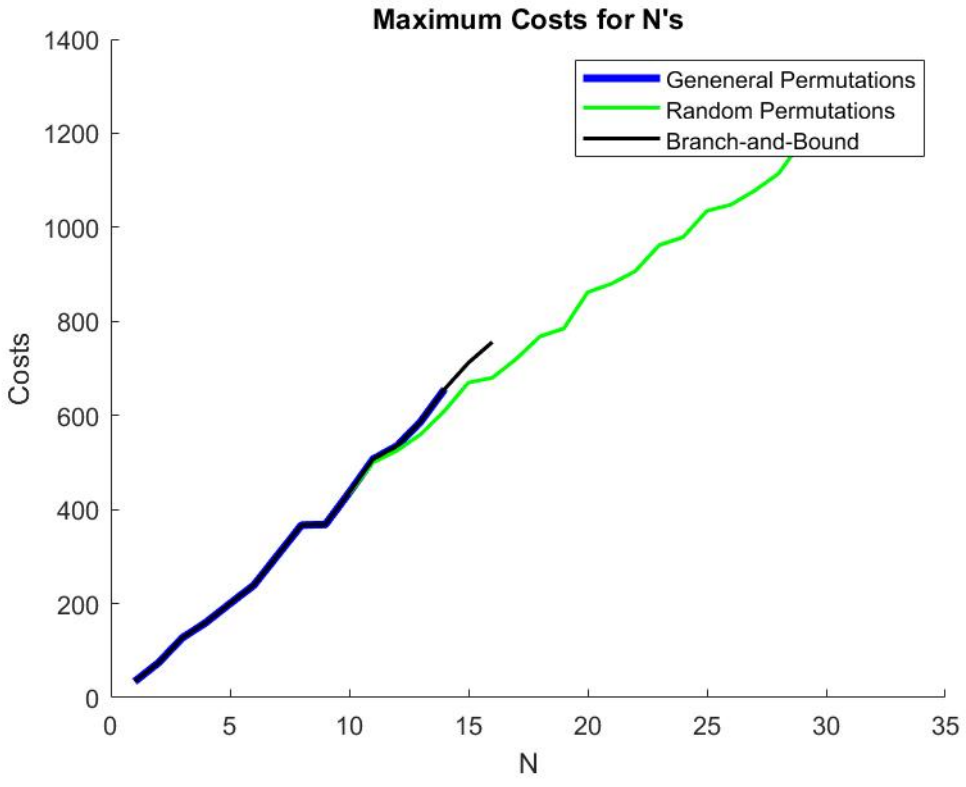
Results Interpretation

As we can see from the graphics bellow, the execution time of Brute Force General Permutations is the slowest one of the other methods because of its complexity O(N*N!), and then it's the Branch-and-Bound method which grows with a complexity, approximately, of O(2^N) and finally the Random Permutations which is the fastest of them all given it's simple complexity of O(N).



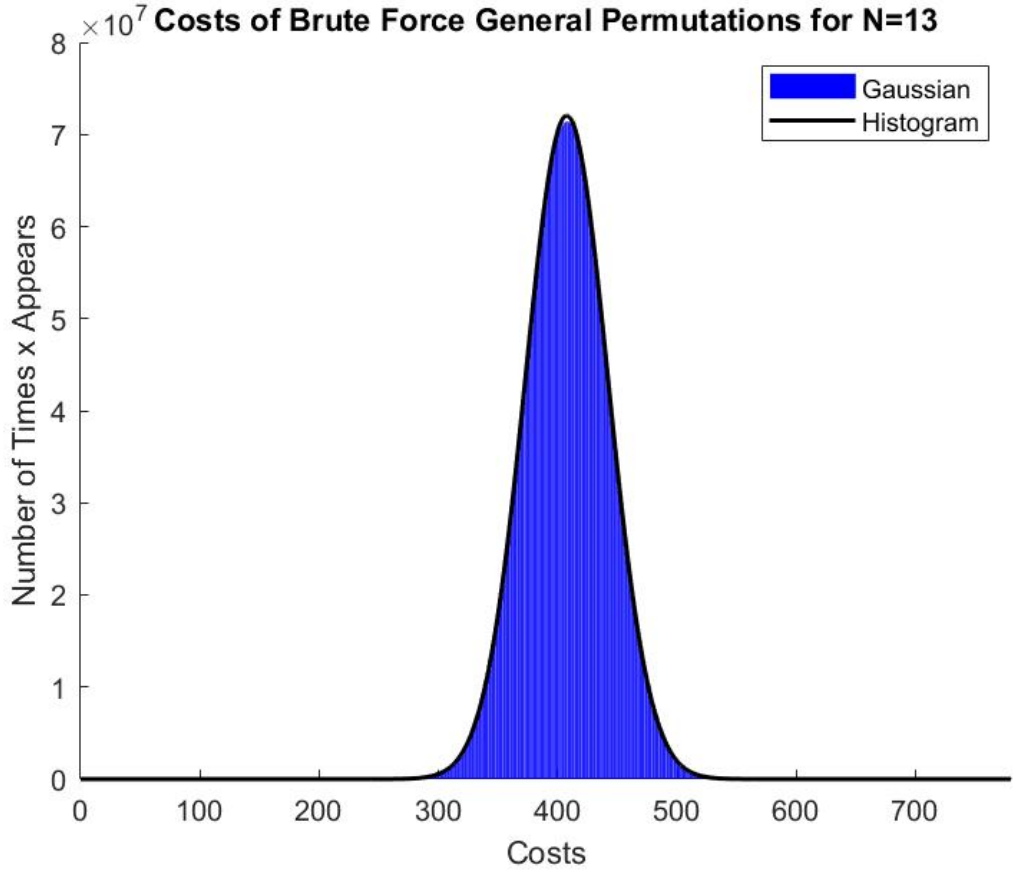
The Random Permutations method is the fastest but at a cost, it's not very accurate because it only does random 1 000 000 permutations it doesn't always get the right assignment for the minimum and maximum cases. The other two methods are 100% accurate therefore the values of the minimum and maximum costs are the same as we can see on the graphic.

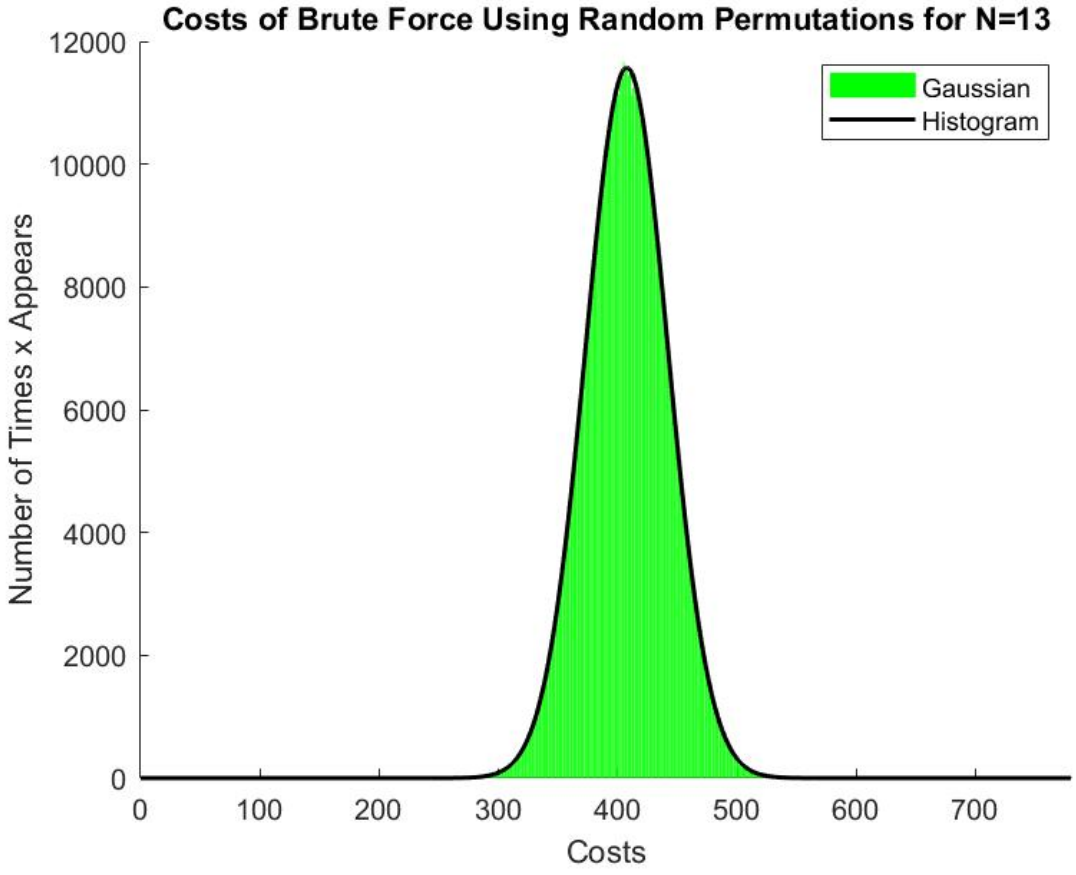

Minimum Costs for N's
Geneneral Permutations
Random Permutations
Branch-and-Bound
Costs
N
0
100
200
300
400
500
600
700
800
0
5
10
15
20
25
30
35

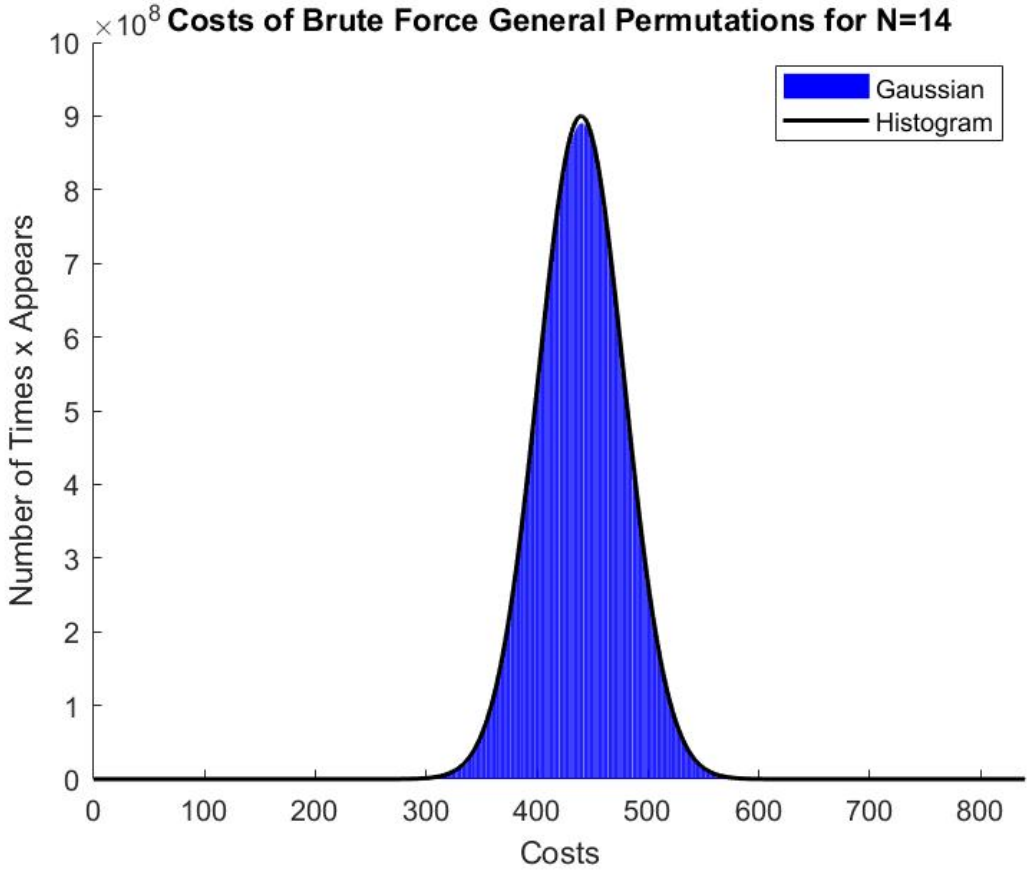

Maximum Costs for N's
Geneneral Permutations
Random Permutations
Branch-and-Bound
Costs
N
0
200
400
600
800
1000
1200
1400
0
5
10
15
20
25
30
35

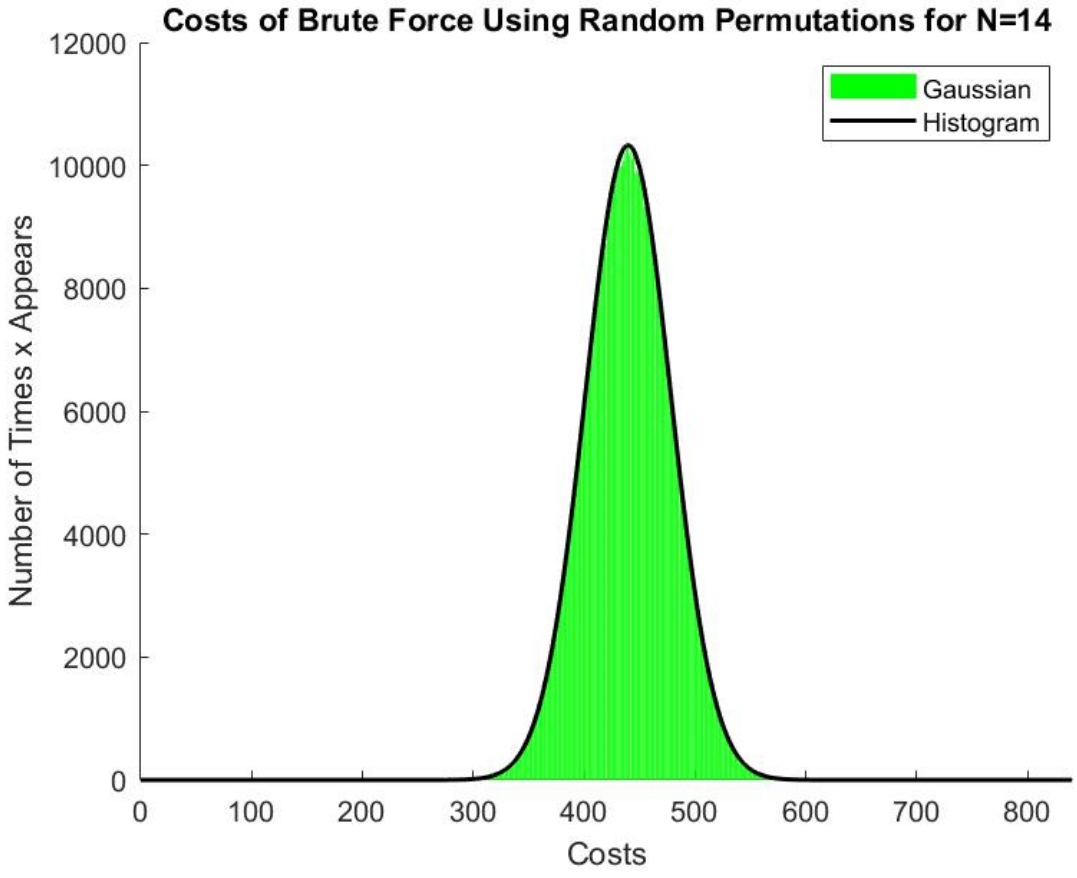
This next graphics show the amount of times a permutation with a certain cost has appeared and it has the highest amount of times on the average of the costs as we can see.

By analysing this next graphics, we got to the conclusion the permutations followed almost a perfect normal distribution by doing the gaussian graph on top of the histogram and realised they are almost identical, however they aren’t exactly equal and especially on the random permutations’ graphics. Also, we concluded that a sample from a normal distribution function will also have a normal distribution because the random permutation is just a sample from the general permutations and it also follows almost a perfect normal distribution.


Costs of Brute Force General Permutations for N=13
×10^7
Gaussian
Histogram
Number of Times x Appears
Costs
0
1
2
3
4
5
6
7
8
0
100
200
300
400
500
600
700


Costs of Brute Force Using Random Permutations for N=13
Gaussian
Histogram
Number of Times x Appears
Costs
0
2000
4000
6000
8000
10000
12000
0
100
200
300
400
500
600
700


Costs of Brute Force General Permutations for N=14
×10^8
Gaussian
Histogram
Number of Times x Appears
Costs
0
1
2
3
4
5
6
7
8
9
10
0
100
200
300
400
500
600
700
800


Costs of Brute Force Using Random Permutations for N=14
Gaussian
Histogram
Number of Times x Appears
Costs
0
2000
4000
6000
8000
10000
12000
0
100
200
300
400
500
600
700
800

Conclusion

With this work we learned how important an algorithm complexity is. It must be considered before making any algorithm according to the programmer preferences or necessities. The big factors that take place in this decision are the fastness and simplicity. Usually, the more fastness, the less simplicity and vice-versa. Thus, sometimes a balance is required.

It is worth of pointing that for small problems, it is preferable to make practical and simple programs as the complexity will have little effect in the time execution.

Referencies

Wikipedia – Assignment Problem: https://en.wikipedia.org/wiki/Assignment_problem

GeeksforGeeks – Hungarian Algorithm: https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/

Appendix

```
        %% Compare Execution Times
        TimeGen=load("TimeGen");
        TimeRan=load("TimeRan");
        TimeBB=load("TimeBB");
        TimeBBMin=load("TimeBBMin");

        figure(1);
        subplot(3, 1,1);
        plot(TimeGen(:,1), log10(TimeGen(:,2)),
'b','LineWidth',2);
        title('Execution Time of Brute Force
General Permutations');
        xlabel("N");
        ylabel("Execution Time log10(sec)");

        subplot(3, 1,2);
        plot(TimeRan(:,1), log10(TimeRan(:,2)),
'b','LineWidth',2);
        title("Execution Time of Brute Force
Random Permutations");
        xlabel("N");
        ylabel("Execution Time log10(sec)");

        subplot(3, 1,3);
        plot(TimeBB(:,1), log10(TimeBB(:,2)),
'b','LineWidth',2);
        title("Execution Time of Branch-and-
Bound");
        xlabel("N");
        ylabel("Execution Time log10(sec)");


        figure(2);
        hold on;
        title("Execution Time of Branch-and-
Bound");
        plot(TimeBBMin(:,1),
log10(TimeBBMin(:,2)), 'g','LineWidth',2);
        plot(TimeBB(:,1), log10(TimeBB(:,2)),
'b','LineWidth',2);
        legend('Branch-and-Bound Before',
'Branch-and-Bound After');
        xlabel("N");
        ylabel("Execution Time log10(sec)");
        hold off;

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %% For N=13 and N=14 graphics

        GenN13=load("GenN13");
        RanN13=load("RanN13");

        GenN14=load("GenN14");
        RanN14=load("RanN14");

        figure(3);
        c=GenN13(:,1);
        n_c=GenN13(:,2);
        med=sum(n_c.*c)/sum(n_c);
        var=sum(n_c.*(c-med).^2)/sum(n_c);
        gaus=(1/sqrt(2*pi*var))*exp(-(c-
med).^2/(2*var))*sum(n_c);
        hold on;
        bar(GenN13(:,1), GenN13(:,2), 'b');
        plot(c, gaus, 'k','LineWidth',1.5);
        legend("Gaussiana", "Histogram");
        hold off;title('Costs of Brute Force
General Permutations for N=13');
        xlabel("Costs");
        ylabel("Number of Times x Appears");

        figure(4)
        c=RanN13(:,1);
        n_c=RanN13(:,2);
        med=sum(n_c.*c)/sum(n_c);
        var=sum(n_c.*(c-med).^2)/sum(n_c);
        gaus=(1/sqrt(2*pi*var))*exp(-(c-
med).^2/(2*var))*sum(n_c);
        hold on;
        bar(RanN13(:,1), RanN13(:,2), 'g');
        plot(c, gaus, 'k','LineWidth',1.5);
        legend("Gaussiana", "Histogram");
        hold off;
        title('Costs of Brute Force Using Random
Permutations for N=13');
        xlabel("Costs");
        ylabel("Number of Times x Appears");
        figure(5);
        c=GenN14(:,1);
        n_c=GenN14(:,2);
        med=sum(n_c.*c)/sum(n_c);
        var=sum(n_c.*(c-med).^2)/sum(n_c);
        gaus=(1/sqrt(2*pi*var))*exp(-(c-
med).^2/(2*var))*sum(n_c);
        hold on;
        bar(GenN14(:,1), GenN14(:,2), 'b');
        plot(c, gaus, 'k','LineWidth',1.5);
        legend("Gaussiana", "Histogram");
        hold off;
        title('Costs of Brute Force General
Permutations for N=14');
        xlabel("Costs");
        ylabel("Number of Times x Appears");

        figure(6);
        c=RanN14(:,1);
        n_c=RanN14(:,2);
        med=sum(n_c.*c)/sum(n_c);
        var=sum(n_c.*(c-med).^2)/sum(n_c);
        gaus=(1/sqrt(2*pi*var))*exp(-(c-
med).^2/(2*var))*sum(n_c);
        hold on;
        bar(RanN14(:,1), RanN14(:,2), 'g');
        plot(c, gaus, 'k','LineWidth',1.5);
        legend("Gaussiana", "Histogram");
        hold off;
        title('Costs of Brute Force Using Random
Permutations for N=14');
        xlabel("Costs");
        ylabel("Number of Times x Appears");

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %% Minimums and maximums

        MaxCostBB=load("MaxCostBB");
        MaxCostRan=load("MaxCostRan");
        MaxCostGen=load("MaxCostGen");
        MinCostGen=load("MinCostGen");
        MinCostRan=load("MinCostRan");
        MinCostBB=load("MinCostBB");

        figure(7);
        hold on;
        plot(MinCostGen(:,1), MinCostGen(:,2),
'b','LineWidth',3);
        plot(MinCostRan(:,1), MinCostRan(:,2),
'g','LineWidth',1.5);
        plot(MinCostBB(:,1), MinCostBB(:,2),
'k','LineWidth',1.5);
        legend("Geneneral Permutations", "Random
Permutations", "Branch-and-Bound");
        hold off;
        title("Minimum Costs for N's");
        ylabel("Costs");
        xlabel("N");

        figure(8);
        hold on;
        plot(MaxCostGen(:,1), MaxCostGen(:,2),
'b','LineWidth',3);
        plot(MaxCostRan(:,1), MaxCostRan(:,2),
'g','LineWidth',1.5);
        plot(MaxCostBB(:,1), MaxCostBB(:,2),
'k','LineWidth',1.5);
        legend("Geneneral Permutations", "Random
Permutations", "Branch-and-Bound");
        hold off;
        title("Maximum Costs for N's");
        ylabel("Costs");
        xlabel("N");
```

```
// AED, 2019/2020
//
// Leandro Emanuel Soares Alves da Silva 93446
// Mário Francisco Silva 93430
//
// Brute-force solution of the assignment problem (https://en.wikipedia.org/wiki/Assignment_problem)
//
// Compile with "cc -Wall -O2 assignment.c -lm" or equivalent
//
// In the assignment problem we will solve here we have n agents and n tasks; assigning agent
//   a
// to task
//   t
// costs
//   cost[a][t]
// The goal of the problem is to assign one agent to each task such that the total cost is minimized
// The total cost is the sum of the costs
//
// Things to do:
//   0. (mandatory)
//      Place the student numbers and names at the top of this file
//   1. (highly recommended)
//      Read and understand this code
//   2. (mandatory)
//      Modify the function generate_all_permutations to solve the assignment problem
//      Compute the best and worst solutions for all problems with sizes n=2,...,14 and for each
//      student number of the group
//   3. (mandatory)
//      Calculate and display an histogram of the number of occurrences of each cost
//      Does it follow approximately a normal distribution?
//      Note that the maximum possible cost is n * t_range
//   4. (optional)
//      For each problem size, and each student number of the group, generate one million (or more!)
//      random permutations and compute the best and worst solutions found in this way; compare
//      these solutions with the ones found in item 2
//      Compare the histogram computed in item 3 with the histogram computed using the random
//      permutations
//   5. (optional)
//      Try to improve the execution time of the program (use the branch-and-bound technique)
//   6. (optional)
//      Surprise us, by doing something more!
//   7. (mandatory)
//      Write a report explaining what you did and presenting your results

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#define NDEBUG  // uncomment to skip disable asserts (makes the code slightly faster)
#include <assert.h>

////////////////////////////////////////////////////////////////////////////////////////
//
// problem data
//
// max_n ..... maximum problem size
// cost[a][t] ... cost of assigning agent a to task t
//

//
// if your compiler complains about srandom() and random(), replace #if 0 by #if 1
//
#if 0
# define srandom srand
# define random  rand
#endif

#define max_n    32          // do not change this (maximum number of agents, and tasks)
#define range    20          // do not change this (for the pseudo-random generation of costs)
#define t_range  (3 * range)  // do not change this (maximum cost of an assignment)

static int cost[max_n][max_n];
static int seed; // place a student number here!

static void init_costs(int n)
{
  if(n == -3)
  { // special case (example for n=3)
    cost[0][0] = 3; cost[0][1] = 8; cost[0][2] = 6;
    cost[1][0] = 4; cost[1][1] = 7; cost[1][2] = 5;
    cost[2][0] = 5; cost[2][1] = 7; cost[2][2] = 5;
    return;
  }
  if(n == -5)
  { // special case (example for n=5)
    cost[0][0] = 27; cost[0][1] = 27; cost[0][2] = 25; cost[0][3] = 41; cost[0][4] = 24;
    cost[1][0] = 28; cost[1][1] = 26; cost[1][2] = 47; cost[1][3] = 38; cost[1][4] = 21;
    cost[2][0] = 22; cost[2][1] = 48; cost[2][2] = 26; cost[2][3] = 14; cost[2][4] = 24;
    cost[3][0] = 32; cost[3][1] = 31; cost[3][2] =  9; cost[3][3] = 41; cost[3][4] = 36;
    cost[4][0] = 24; cost[4][1] = 34; cost[4][2] = 30; cost[4][3] = 35; cost[4][4] = 45;
    return;
  }
  assert(n >= 1 && n <= max_n);
  srandom((unsigned int)seed * (unsigned int)max_n + (unsigned int)n);
  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]
}

////////////////////////////////////////////////////////////////////////////////////////
//
// code to measure the elapsed time used by a program fragment (an almost copy of elapsed_time.h)
//
// use as follows:
//
```

```
//   (void)elapsed_time();
//   // put your code to be time measured here
//   dt = elapsed_time();
//   // put morecode to be time measured here
//   dt = elapsed_time();
//
// elapsed_time() measures the CPU time between consecutive calls
//

#if defined(__linux__) || defined(__APPLE__)
//
// GNU/Linux and MacOS code to measure elapsed time
//
#include <time.h>

static double elapsed_time(void)
{
   static struct timespec last_time,current_time;

   last_time = current_time;
   if(clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&current_time) != 0)
      return -1.0; // clock_gettime() failed!!!
   return          ((double)current_time.tv_sec - (double)last_time.tv_sec)
          + 1.0e-9 * ((double)current_time.tv_nsec - (double)last_time.tv_nsec);
}

#endif

#if defined(_MSC_VER) || defined(_WIN32) || defined(_WIN64)

//
// Microsoft Windows code to measure elapsed time
//

#include <windows.h>

static double elapsed_time(void)
{
   static LARGE_INTEGER frequency,last_time,current_time;
   static int first_time = 1;

   if(first_time != 0)
   {
      QueryPerformanceFrequency(&frequency);
      first_time = 0;
   }
   last_time = current_time;
   QueryPerformanceCounter(&current_time);
   return (double)(current_time.QuadPart - last_time.QuadPart) / (double)frequency.
QuadPart;
}

#endif

//
// function to generate a pseudo-random permutation
//

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_M
AX)); // range 0..i
      assert(j >= 0 && j <= i);
      int k = t[i];
      t[i] = t[j];
      t[j] = k;

   }
}

// place to store best and worst solutions (also code to print them)
//

static int min_cost,min_cost_assignment[max_n], min_cost_row[max_n]; // smallest
cost information
static int max_cost,max_cost_assignment[max_n], max_cost_row[max_n]; // largest
cost information
static long n_visited; // number of permutations visited (examined)
// place your histogram global variable here
static double cpu_time;
static int histogram[max_n*t_range] = { 0 };

#define minus_inf  -1000000000  // a very small integer
#define plus_inf   +1000000000  // a very large integer

static void reset_solutions(void)
{
   min_cost = plus_inf;
   max_cost = minus_inf;
   n_visited = 0l;
   // place your histogram initialization code here
   for( int i = 0; i < 60*max_n; i++)
      histogram[i] = 0;
   cpu_time = 0.0;
}

#define show_info_1       (1 << 0)
#define show_info_2       (1 << 1)
#define show_costs        (1 << 2)
#define show_min_solution  (1 << 3)
#define show_max_solution  (1 << 4)
#define show_histogram     (1 << 5)
#define show_all          (0xFFFF)

double factorial(int n)
{
  double c;
  double result = 1;

  for (c = 1; c <= n; c++)
    result = result * c;

  return result;
}
```

```
static void show_solutions(int n,char *header,int what_to_show, int type)
{
  printf("%s\n",header);
  if((what_to_show & show_info_1) != 0)
  {
    printf("  seed ......... %d\n",seed);
    printf("  n ............. %d\n",n);
  }
  if((what_to_show & show_info_2) != 0)
  {
    printf("  visited ....... %ld\n",n_visited);
    printf("  cpu time ...... %.3fs\n",cpu_time);
  }
  if((what_to_show & show_costs) != 0)
  {
    printf("  costs .........");
    for(int a = 0;a < n;a++)
    {
      for(int t = 0;t < n;t++)
        printf(" %2d",cost[a][t]);
      printf("\n%s",(a < n - 1) ? "                " : "");
    }
  }
  if((what_to_show & show_min_solution) != 0)
  {
    printf("  min cost ...... %d\n",min_cost);
    if(min_cost != plus_inf)
    {
      printf("  assignement ...");
      for(int i = 0;i < n;i++)
        printf(" %d",min_cost_assignment[i]);
      printf("\n");
    }
  }
  if((what_to_show & show_max_solution) != 0)
  {
    printf("  max cost ...... %d\n",max_cost);
    if(max_cost != minus_inf)
    {
      printf("  assignement ...");
      for(int i = 0;i < n;i++)
        printf(" %d",max_cost_assignment[i]);
      printf("\n");
    }
  }
  if((what_to_show & show_histogram) != 0)
  {
    // place your code to print the histogram here
    FILE *fp;
    char name[60];
    char number[20];

    if (type==0){ // general permutations histogram
      strcpy(name, "GenN");
      snprintf(number, sizeof(number), "%d", n);
      strcat(name, number);
    } else if (type==1){ // random permutations histogram
      strcpy(name, "RanN");
      snprintf(number, sizeof(number), "%d", n);
      strcat(name, number);
    }

    fp=fopen(name, "w+");
    for(int i=0; i<=n*t_range;i++){
      fprintf(fp, "%d %d\n", i, histogram[i]);
    }
    if (type==1 || type == 0) {
      fclose(fp);
    }
  }
}

//
// code used to generate all permutations of n objects
//
// n ..... number of objects
// m ..... index where changes occur (a[0], ..., a[m-1] will not be changed)
// a[idx] ... the number of the object placed in position idx
//
// TODO: modify the following function to solve the assignment problem
//

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);                    // exchange a[i] with a[m]
      generate_all_permutations(n,m + 1,a); // recurse
      swap(i,m);                    // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else
  {
    n_visited++;
    int total_cost = 0;
    for( int i=0; i<n;i++){
      total_cost += cost[i][a[i]];
    }
    if (total_cost<min_cost){
      min_cost=total_cost;
      for(int j = 0; j < n; j++)
        min_cost_assignment[j] = a[j];
    }
    if (total_cost>max_cost){
      max_cost=total_cost;
      for(int j = 0; j < n; j++)
        max_cost_assignment[j] = a[j];
    }

    histogram[total_cost] += 1;
  }
}
//
// code to generate min cost using branch-and-bound method of n objects
```

```
//
// n ..... number of objects
// m ..... index where changes occur (a[0], ..., a[m-1] will not be changed)
// a[idx] ... the number of the object placed in position idx
// parcial_cost ..... parcial cost

static void costs_row(int n)
{
  int min, max;
  for(int a = 0;a < n; a++){
    min = plus_inf; max = minus_inf;
    for(int t = 0; t < n; t++){
      if(min > cost[a][t])
        min = cost[a][t];
      if(max < cost[a][t])
        max = cost[a][t];
    }
    min_cost_row[a] = min;
    max_cost_row[a] = max;
  }
}

static void min_branch_and_bound(int n, int m,int a[n], int partial_cost)
{
  int min = 0;
  for(int i = m; i < n; i++){ min += min_cost_row[i]; }

  if(min_cost < partial_cost + min)
    return;
  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);                    // exchange a[i] with a[m]
      min_branch_and_bound(n,m + 1,a, partial_cost + cost[m][a[m]]); // recurse
      swap(i,m);                    // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else
  {
    int total_cost = partial_cost + cost[m][a[m]];
    if(min_cost > total_cost){
      min_cost = total_cost;
      for(int j = 0; j < n; j++)
        min_cost_assignment[j] = a[j];
    }
    n_visited++;
  }
}
static void max_branch_and_bound(int n, int m,int a[n], int partial_cost)
{
  int max = 0;
  for(int i = m; i < n; i++){ max += max_cost_row[i]; }

  if(max_cost > partial_cost + max)
    return;
  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);                    // exchange a[i] with a[m]
      max_branch_and_bound(n,m + 1,a, partial_cost + cost[m][a[m]]); // recurse
      swap(i,m);                    // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else
  {
    int total_cost = partial_cost + cost[m][a[m]];
    if(max_cost < total_cost){
      max_cost = total_cost;
      for(int j = 0; j < n; j++)
        max_cost_assignment[j] = a[j];
    }
    n_visited++;
  }
}

static void generate_all_permutations_branch_and_bound(int n, int m,int a[n], int par
tial_cost)
{
  costs_row(n);
  min_branch_and_bound(n, m, a, partial_cost);
  max_branch_and_bound(n, m, a, partial_cost);
}

//
// main program
//
int main(int argc,char **argv)
{
  if(argc == 2 && argv[1][0] == '-' && argv[1][1] == 'e')
  {
    seed = 0;
    {
      int n = 3;
      init_costs(-3); // costs for the example with n = 3
      int a[n];
      for(int i = 0;i < n;i++)
        a[i] = i;
      reset_solutions();
      (void)elapsed_time();
      generate_all_permutations(n,0,a);
      cpu_time = elapsed_time();
      show_solutions(n,"Example for n=3",show_all, 0);
      printf("\n");
    }
    {
      int n = 5;
      init_costs(-5); // costs for the example with n = 5
      int a[n];
      for(int i = 0;i < n;i++)
        a[i] = i;
      reset_solutions();
      (void)elapsed_time();
```

```
                generate_all_permutations(n,0,a);
                cpu_time = elapsed_time();
                show_solutions(n,"Example for n=5",show_all, 0);
                return 0;
              }
            }
            if(argc == 2)
            {
              seed = atoi(argv[1]); // seed = student number
              if(seed >= 0 && seed <= 1000000)
              {
                for(int n = 1;n <= max_n;n++)
                {
                  init_costs(n);
                  show_solutions(n,"-------------------------------------------------------------------------\nProblem statement",show_info_1 | show_costs, 0);
                  // 2.
                  if(n <= 14) // use a smaller limit here while developing your code
                  {
                    FILE *fp;
                    FILE *map;
                    FILE *mip;
                    int a[n];
                    fp=fopen("TimeGen", "a");
                    mip=fopen("MinCostGen", "a");
                    map=fopen("MaxCostGen", "a");
                    for(int i = 0;i < n;i++)
                      a[i] = i; // initial permutation
                    reset_solutions();
                    (void)elapsed_time();
                    generate_all_permutations(n,0,a);
                    cpu_time = elapsed_time();
                    fprintf(fp, "%d %f\n", n, cpu_time);
                    fclose(fp);
                    fprintf(mip, "%d %d\n", n, min_cost);
                    fclose(mip);
                    fprintf(map, "%d %d\n", n, max_cost);
                    fclose(map);
                    show_solutions(n,"Brute force",show_info_2 | show_min_solution | show_max_solution | show_histogram,0);
                  }
                  if(n <= 16)
                  {
                    FILE *fp;
                    FILE *map;
                    FILE *mip;
                    int a[n];
                    fp=fopen("TimeBB", "a");
                    mip=fopen("MinCostBB", "a");
                    map=fopen("MaxCostBB", "a");
                    for(int i = 0;i < n;i++)
                      a[i] = i; // initial permutation
                    reset_solutions();
                    (void)elapsed_time();
                    generate_all_permutations_branch_and_bound(n,0,a,0);
                    cpu_time = elapsed_time();
                    fprintf(fp, "%d %f\n", n, cpu_time);
                    fclose(fp);
                    fprintf(mip, "%d %d\n", n, min_cost);
                    fclose(mip);
                    fprintf(map, "%d %d\n", n, max_cost);
                    fclose(map);
                    show_solutions(n,"\nBrute force with branch-and-bound",show_info_2 | show_min_solution | show_max_solution, 2);
                  }
                  FILE *fp;
                  FILE *map;
                  FILE *mip;
                  int t[n];
                  fp=fopen("TimeRan", "a");
                  mip=fopen("MinCostRan", "a");
                  map=fopen("MaxCostRan", "a");
                  reset_solutions();
                  (void)elapsed_time();
                  for(int i=0;i<1000000;i++)
                  {
                    random_permutation(n, t);
                    int total_cost = 0;
                    for( int i=0; i<n;i++){
                      total_cost += cost[i][t[i]];
                    }
                    if (total_cost<min_cost){
                      min_cost=total_cost;
                      for(int j = 0; j < n; j++)
                        min_cost_assignment[j] = t[j];
                    }
                    if (total_cost>max_cost){
                      max_cost=total_cost;
                      for(int j = 0; j < n; j++)
                        max_cost_assignment[j] = t[j];
                    }
                    histogram[total_cost] += 1;
                  }
                  cpu_time = elapsed_time();
                  fprintf(fp, "%d %f\n", n, cpu_time);
                  fprintf(mip, "%d %d\n", n, min_cost);
                  fclose(mip);
                  fprintf(map, "%d %d\n", n, max_cost);
                  fclose(map);
                  n_visited=1000000;
                  show_solutions(n,"\nRandom permutations",show_info_2 | show_min_solution | show_max_solution | show_histogram,1);
                  fclose(fp);
                  // done
                  printf("\n");
                }
                return 0;
              }
            }
            fprintf(stderr,"usage: %s -e          # for the examples\n",argv[0]);
            fprintf(stderr,"usage: %s student_number\n",argv[0]);
            return 1;
          }
```