Algoritmos e Estruturas de Dados

Primeiro Trabalho Prático

The Assignment Problem

9 de Novembro de 2019

Trabalho Realizado por:

- Daniel Gomes, Nmec 93015, Contributo:50%
- Rui Fernandes, Nmec 92952, Contributo: 50%

Índice

1. Introdução

1.1. O que é o Assignment Problem?

O objetivo do primeiro trabalho prático foi estudar e analisar o "*Assignment Problem".* Neste problema temos por base um conjunto de agentes e outro de tarefas, no caso específico do nosso estudo analisamos o "*Balanced Assignment Problem"* implicando que os conjuntos têm o mesmo número de elementos, denominado por ***n***.

Atribuir um agente a uma tarefa tem um determinado custo, isto para cada uma das combinações agente-tarefa. Tendo em conta que cada tarefa só pode ser atribuída uma vez, o objetivo do problema é atribuir um agente a cada tarefa de tal modo que o custo total (soma dos custos de cada atribuição) seja o mínimo possível.

2. Métodos Utilizados

2.1. Brute Force

Um método de simples implementação do problema, denominado "*Brute Force",* baseia-se em percorrer todas as permutações possíveis de agentes-tarefas e encontrar a que corresponde ao custo total mínimo (sendo também fácil encontrar o custo total máximo).

Pode-se definir em 4 passos o algoritmo genérico de *"Brute Force"*:

"

1. *primeiro* (*P*): gera o primeiro candidato à solução de *P*.
2. *próximo* (*P*, *c*): gera o próximo candidato de *P* depois de *c*.
3. *válido* (*P*, *c*): verifica-se o candidato *c* é a solução de *P*.
4. *saída* (*P*, *c*): usa a solução *c* de *P* como for conveniente para a aplicação. "

Citado de
https://pt.wikipedia.org/wiki/Busca_por_força_bruta

Este método tem o benefício de ser muito fácil de implementar em código e chegar sempre ao valor certo, no entanto é extremamente demoroso, tendo uma complexidade algorítmica de O(n*n!).

2.2. Branch and Bound

O *"Branch and Bound Algorithm"* é uma técnica que pode ser aplicada a diversos tipos de problemas nomeadamente na área de otimização combinatória, onde o problema em causa, *"Assignment Problem"*, é um exemplo.

Este método assemelha-se a princípios que caracterizam o *Brute Force* mas com a ligeira diferença de que não são analisados os casos que, à partida, já são do nosso conhecimento que nunca poderão ser solução. Assim, são, portanto, casos descartáveis para solucionar o problema em causa, pelo que apenas consideramos as opções que têm a possibilidade de serem candidatas a ser a solução ou que preencham alguns requisitos iniciais necessários para tal.

Ao contrário do *"Brute Force"*, o "*Branch and Bound"* não apresenta uma complexidade computacional definida, segundo os *computer scientists.* Isto acontece porque calcular a complexidade deste algoritmo é um processo extremamente difícil e que pode ser variável dependendo da sua aplicação.

2.3. Random Permutations

Um método diferente de resolver o problema é ao invés de percorrer todas as permutações possíveis, gerar um número fixo (no nosso estudo utilizámos 1e6) de permutações aleatórias e calcular o custo mínimo utilizando as mesmas.

Apesar deste método ter uma complexidade algorítmica extremamente menor (ou seja, é muito menos demoroso), não garante que se encontre o verdadeiro custo mínimo, apenas próximo.

Quando o número de permutações aleatórias é consideravelmente maior que o número total de permutações possíveis é muito provável o valor calculado ser o custo mínimo real e, pelo contrário, quanto menor for mais distante fica o valor calculado do valor real.

É de notar que se se aumentar este valor fixo de permutações para igualar o número total, torna-se mais demoroso e menos eficiente criar e analisar mais permutações.

2.4. Outros Métodos e Notas Interessantes

Por falta de tempo e nalguns casos muita dificuldade de implementação ou teste, não testámos os seguintes métodos e soluções do *"Assignment Problem"*, no entanto achámos importante referi-los neste relatório.

Um dos métodos mais eficiente a resolver o *"Assignment Problem",* com uma complexidade algorítmica de O(n^3), é o *"Hungarian Algorithm"* desenvolvido por Harold Khun.

O algoritmo assenta nos seguintes passos:

1. For each row of the matrix, 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.

Citado de
https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/

Um teste muito interessante de efetuar seria o Teste de Kolmogorov-Smirnov, onde a estatística utilizada para o teste é :

$$D_n = \sup |F_n(x) - F(x)|$$

- $F(x)$ representa a função de distribuição acumulada assumida para os dados;
- $F_n(x)$ representa a função de distribuição acumulada empírica dos dados.

onde:

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{[-\infty,x]}(X_i)$$

e em que ***sup*** é o supremo do conjunto de distâncias.

Este teste observa a máxima diferença absoluta entre a função de distribuição acumulada assumida para os dados, e a função de distribuição empírica dos dados. Como critério, comparamos esta diferença com um valor crítico, para um dado nível de significância.

Adaptado de
http://www.portalaction.com.br/inferencia/62-teste-de-kolmogorov-smirnov

Por fim, é importante mencionar que o Assignment Problem pode ser associado a *Tropical Geometry* ou *Max-plus Algebra,* onde o custo mínimo total para cada caso é de facto o determinante trópico de uma matriz ***X***(***n*** por ***n***) :

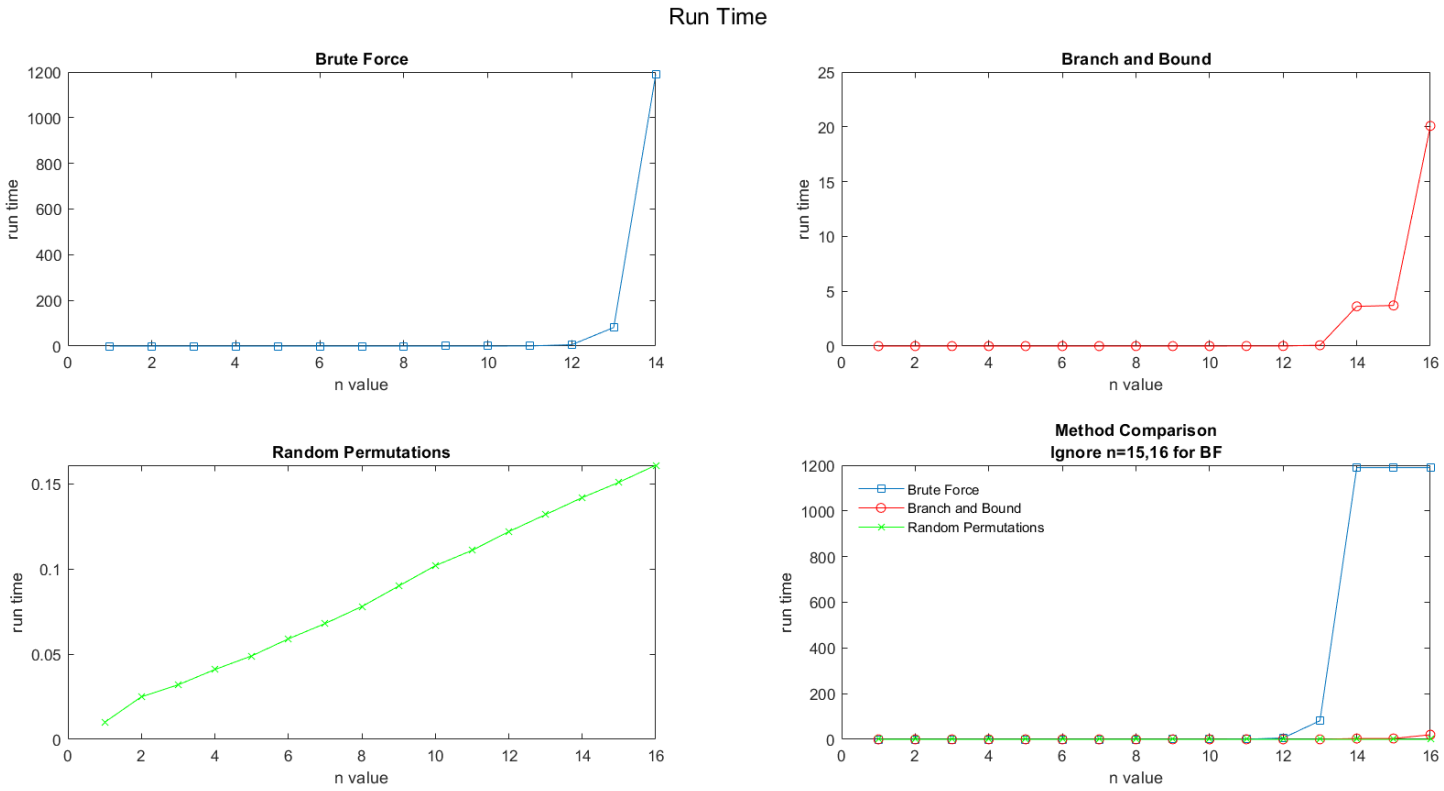
$$\operatorname{tropdet}(X) \quad := \quad \bigoplus_{\pi \in S_n} x_{1\pi(1)} \odot x_{2\pi(2)} \odot \cdots \odot x_{n\pi(n)}.$$

Adaptado de https://personal-homepages.mis.mpg.de

3. Resultados Obtidos

3.1. Comparação de eficiência dos métodos

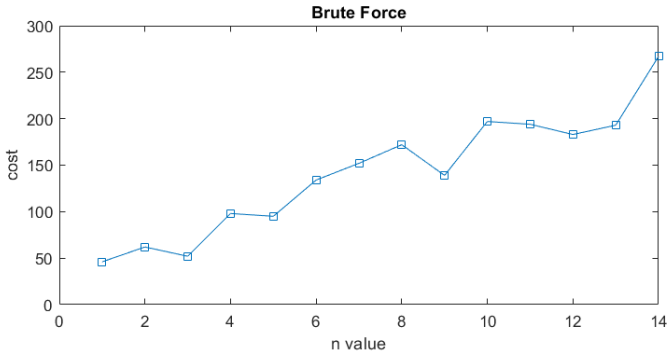
Deve-se notar que os seguintes resultados de Run Time e dos custos mínimos e máximos foram efetuados usando a seed 92952.

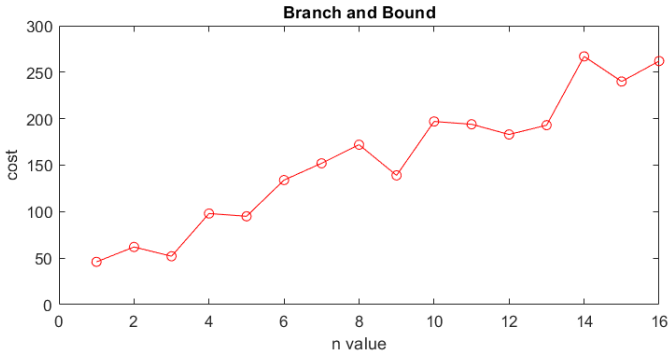


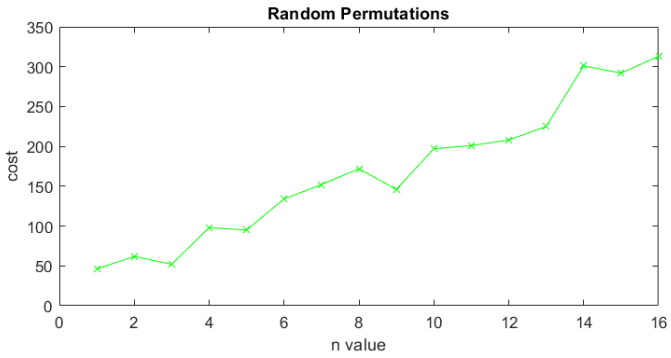
Como é possível observar nos gráficos de *run time*(s), confirma-se que o método de *"Brute Force"* é de longe o que se torna mais demorado com o aumento de ***n***. Para ***n***=14 já tem um *run time* de cerca de 1200s, ou seja, 20 minutos, enquanto que o *"Branch and Bound"* demora menos de 5 segundos. Confirma-se ainda que o gráfico do *"Brute Force"* tem de facto a forma aproximada de **n*n!.**

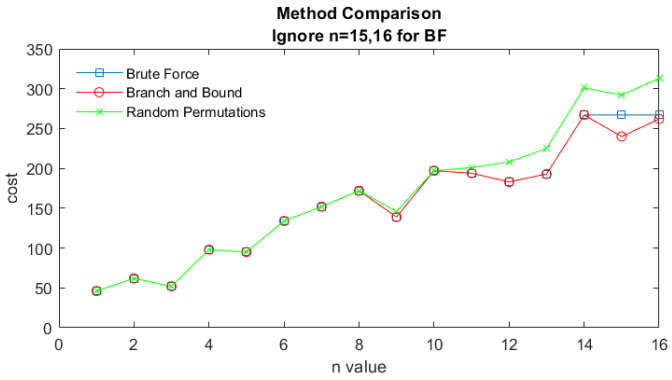
Analisando o gráfico das *"Random Permutations"* denota-se que se assemelha em forma com uma função de ordem n, e é como esperado a que se mantem com um *run time* mais baixo á medida que o valor de ***n*** aumenta.

Min Cost

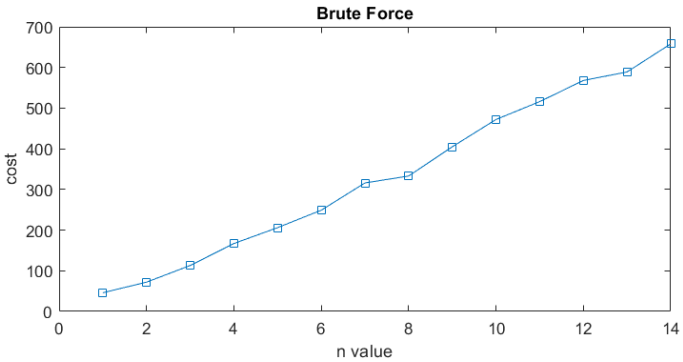


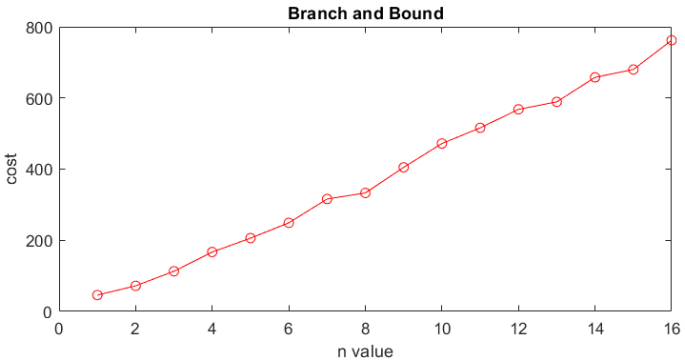


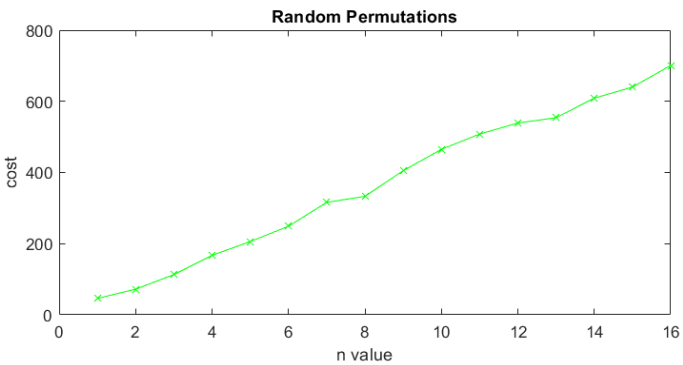


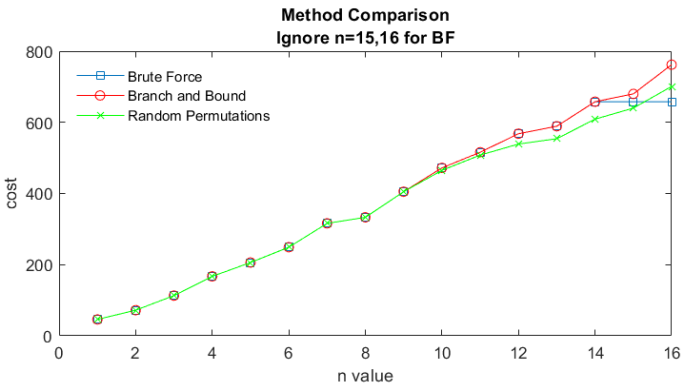


Max Cost





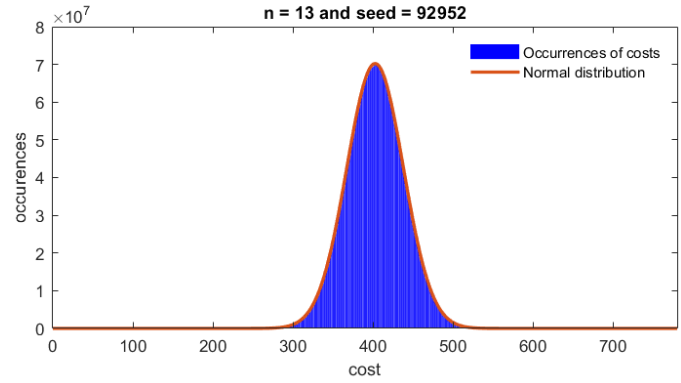


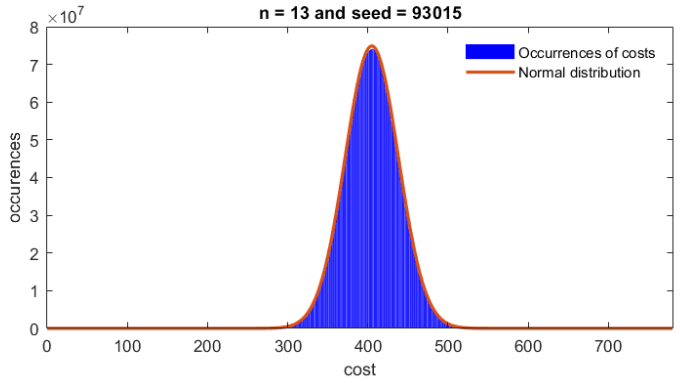


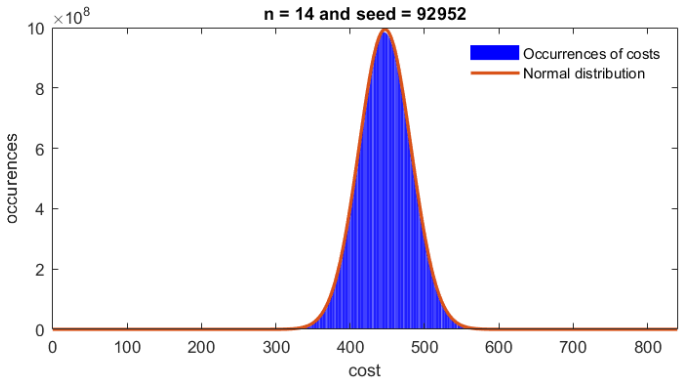
Nos gráficos acima, estão representados os custos mínimos e máximos calculados pelos diversos métodos com o crescer de ***n***. Confirma-se que sendo os valores encontrados pelo *"Brute Force"* os verdadeiros, o *"Branch and Bound"* encontra precisamente os mesmos valores. Acontece ainda o afastamento dos valores encontrados pelas *"Random Permutations"* dos valores reais "fugindo" a estes cada vez mais á medida que o ***n*** aumenta.

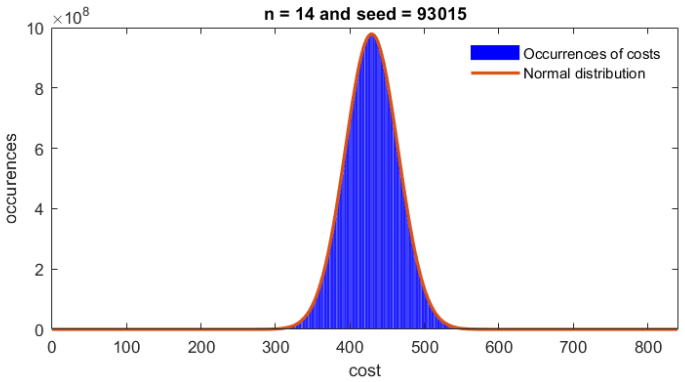
3.2. Análise de Histogramas

BruteForce



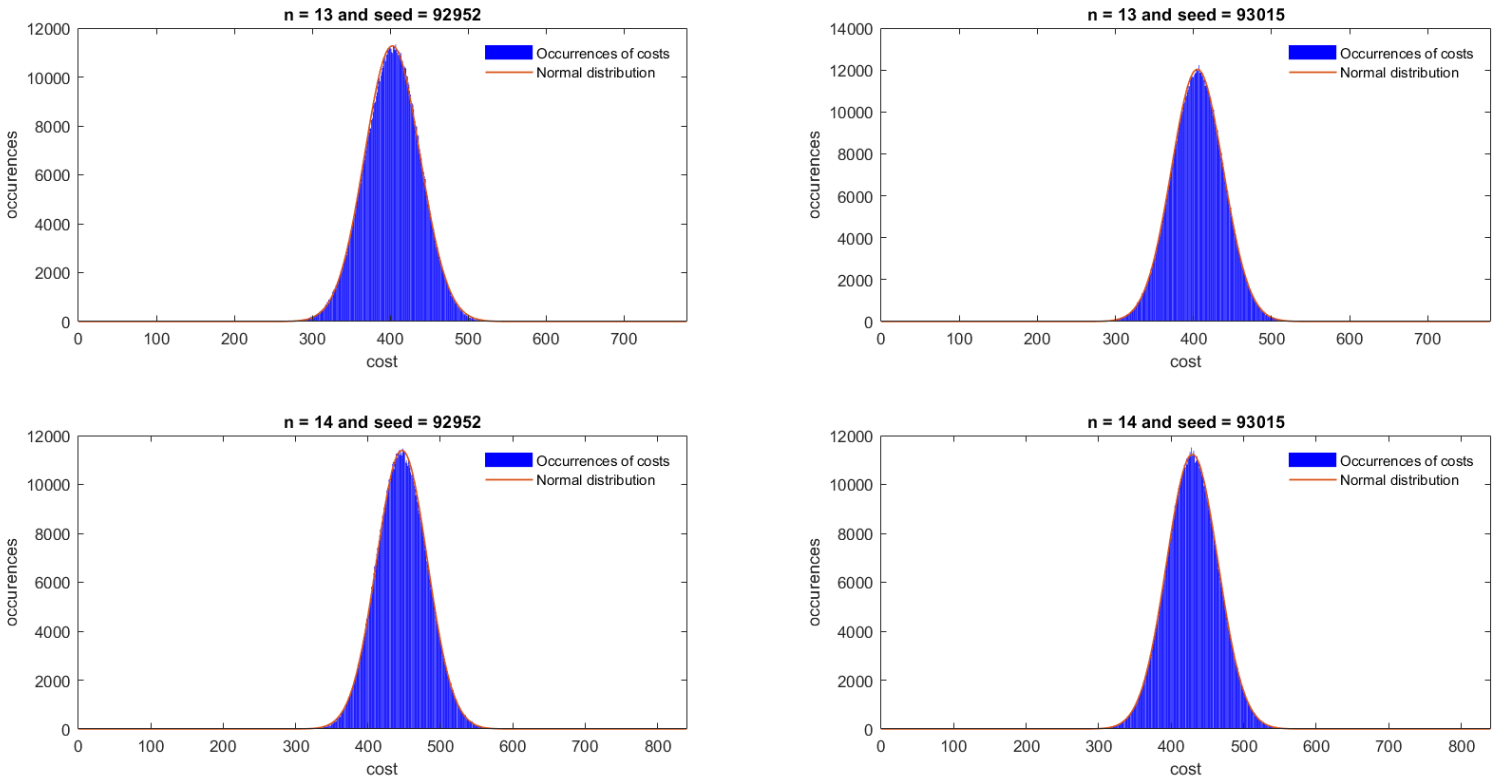






Um gráfico interessante de analisar é o histograma do número de ocorrências de cada custo possível para um dado n (neste caso quando igual a 13 e 14). Principalmente para valores de n superior onde existe um numero considerável de permutações, o histograma tende para seguir a forma de uma distribuição normal, praticamente coincidindo com a mesma.

Random Permutations



No caso das *"Random Permutations"* observa-se algo semelhante exceto que a forma se torna menos uniforme, "fugindo" mais á Gaussiana, isto era expectável visto que se trata de uma amostra muito menor de permutações ao invés de todas as hipóteses.

4. Código Desenvolvido

4.1. Assignment.c

A maior parte deste código foi nos disponibilizado numa das aulas práticas ,sendo a nossa base do projeto, e cabendo a nós completar a função *generate_all_permutations* (de forma a obter o custo máximo e mínimo para cada n e guardar os dados para conseguir obter histogramas), assim como criar a função *generate_all_permutations_branch_bound* para implementação do *branch and bound,* e chamar a função *random_permutation* que já tinha sido disponibilizada *.* Como forma de compactar o código foi criada a função *evaluate_permutation*.Além disto foi gerada a função *export_histogram* como forma de poderem ser exportados, para o *Matlab,* os dados dos histogramas e gráficos analisados anteriormente neste Relatório. Por fim, para que fosse possível gerar automaticamente os histogramas pretendidos assim que fosse corrido o código recorremos ao *GnuPlot (*função *plot_histogram*).

```c
#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 FILE *gnuplot;

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)
```

```
  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_MAX)); // 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]; // smallest cost information
static int max_cost,max_cost_assignment[max_n]; // largest cost information
static long n_visited; // number of permutations visited (examined)
static int histogram[max_n*t_range +1]; // place your histogram global variable here
static double cpu_time;

#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;
```

```
  min_cost = plus_inf;
  max_cost = minus_inf;
  n_visited = 0l;
  memset(histogram, 0, (max_n*t_range +1)*sizeof(histogram[0])); // place your histogram initialization code here
  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)

static void export_histogram(int hist[], int n, int seed){

  //exports to a txt file so the values can be imported in to matlab, can be altered to make a dat file for gnuplot

  char n_str[6];
  sprintf(n_str, "%d", n);
  char seed_str[6];
  sprintf(seed_str, "%d", seed);

  char fname[80];
  strcpy(fname, "histogram_");
  strcat(fname, n_str);
  strcat(fname, "_");
  strcat(fname, seed_str);
  strcat(fname, ".txt");

  FILE *fp = fopen(fname, "w");
  for(int i=0;i<(n*t_range); i++){
    fprintf(fp,"%d,",hist[i]);
  }
  fprintf(fp,"%d",hist[n*t_range]);
  fclose(fp);
}

static void plot_histogram(int hist[], int n, int seed, char *method){

  gnuplot = popen("gnuplot -persist","w");

  fprintf(gnuplot, "set term qt title '%s'\n",method);
  fprintf(gnuplot, "set style data histograms\n");
  fprintf(gnuplot, "set style histogram clustered gap 1.2\n");
  fprintf(gnuplot, "set boxwidth 0.8 relative\n");
  fprintf(gnuplot, "set style fill solid\n");
  fprintf(gnuplot, "plot '-' using 2 linecolor 'red' title 'Occurrences of Costs for n=%d and seed=%d'\n",n,seed);
  for(int i = min_cost; i <= max_cost; i++){
    fprintf(gnuplot, "%d %d\n",i,hist[i]);
  }
  fprintf(gnuplot,"e");
  fflush(gnuplot);
  pclose(gnuplot);
}

static void show_solutions(int n,char *header,int what_to_show)
{
  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)
  {
    export_histogram(histogram, n, seed);          //uncomment to export histogram
    plot_histogram(histogram, n, seed, header);
    printf("  histogram ...\n");              //  uncomment to print histogram values on the terminal
    for(int i = min_cost; i <= max_cost; i++)
    printf("                [%d]  %d\n",i, histogram[i]);
    printf("\n");
  }
}


////////////////////////////////////////////////////////////////////////////////////////////////////
//
// 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 evaluate_permutation(int n, int a[n])
{
  n_visited++;

    int perm_cost = 0;
    for(int i=0; i < n; i++)
      perm_cost += cost[i][a[i]];

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

```
    if(perm_cost > max_cost)
    {
      max_cost = perm_cost;
      for(int i = 0; i < n; i++)
        max_cost_assignment[i] = a[i];
    }
    histogram[perm_cost] += 1;
}

static void generate_all_permutations(int n,int m,int a[n])
{

  if(m < n - 1)
  {
    //
    // not yet at the end; try all possibilities for a[m]
    //
    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
  {

    evaluate_permutation(n,a);
    // place your code to update the best and worst solutions, and to update the histogram here
  }
}

static void generate_all_permutations_branch_and_bound(int n, int m,int a[n], int parcial_cost)
{
  if(min_cost < parcial_cost + 3*(n - m))
    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]
      generate_all_permutations_branch_and_bound(n,m + 1,a, parcial_cost + cost[m][a[m]]); // recurse
      swap(i,m);                                // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else
  {
    n_visited++;
   int total_cost = parcial_cost+cost[m][a[m]];

    if(min_cost > total_cost){
      min_cost = total_cost;
      for(int i = 0; i < n; i++)
        min_cost_assignment[i] = a[i];
    }

  }
}

static void max_cost_branch_and_bound(int n, int m,int a[n], int parcial_cost)
{
  if(max_cost > parcial_cost + 60*(n - m )) //minimum cost of each cell of the cost table is 3 *(n-m)
```

```
    return ; // pass to next m

  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_cost_branch_and_bound(n,m + 1,a, parcial_cost + cost[m][a[m]]); // recurse
      swap(i,m);                                  // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else
  {
    int total_cost = parcial_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++;
  }
}


////////////////////////////////////////////////////////////////////////////////
//
// 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);
      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);
      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,"Problem statement",show_info_1 | show_costs);
      //
      // 2.
      //
      if(n <= 14)     //Brute Force Method
      {
        int a[n];
        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();
        if(n<=10){    //chose values of n to display histogram
          show_solutions(n,"Brute force",show_info_2 | show_min_solution | show_max_solution);
        } else{
          show_solutions(n,"Brute force",show_info_2 | show_min_solution | show_max_solution | show_histogram);
        }
      }
#if 1
      if(n <= 16)     //Branch and Bound Method
      {
        int a[n];
        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();
        show_solutions(n,"Brute force with branch-and-bound",show_info_2 | show_min_solution);
      }
#endif
       if(n<=16){
        int a[n];
        for(int i = 0;i < n;i++)
          a[i] = i; // initial permutation
        reset_solutions();
        (void)elapsed_time();
        max_cost_branch_and_bound(n,0,a,0);
        cpu_time = elapsed_time();
        show_solutions(n,"Max cost with branch-and-bound",show_info_2 | show_max_solution);
      }


      if (n <=15)     //Random Permutations Method
      {
        int a[n];
        reset_solutions();
        (void)elapsed_time();
        for (int i =0; i<1000000;i++)
        {
          random_permutation(n,a);
          evaluate_permutation(n,a);
        }
        cpu_time = elapsed_time();
        if(n<=10){    //chose values of n to display histogram
            show_solutions(n,"Brute force with random permutations",show_info_2 | show_min_solution | show_max_solution);
          } else{
            show_solutions(n,"Brute force with random permutations",show_info_2 | show_min_solution | show_max_solution | show_histogram);
```

```
        max_cost_branch_and_bound(n,0,a,0);
        cpu_time = elapsed_time();
        show_solutions(n,"Max cost with branch-and-bound",show_info_2 | show_max_solution);
      }


      if (n <=15)     //Random Permutations Method
      {
        int a[n];
        reset_solutions();
        (void)elapsed_time();
        for (int i =0; i<1000000;i++)
        {
          random_permutation(n,a);
          evaluate_permutation(n,a);
        }
        cpu_time = elapsed_time();
        if(n<=10){    //chose values of n to display histogram
            show_solutions(n,"Brute force with random permutations",show_info_2 | show_min_solution | show_max_solution);
          } else{
            show_solutions(n,"Brute force with random permutations",show_info_2 | show_min_solution | show_max_solution | show_histogram);
          }

    }


      //
      // 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;
}
```

4.2. Código Matlab

```
function make_hist(n,seed)

filename = "histogram_"+n+"_"+seed+".txt";

Y = importdata(filename, ',');
X = 0:(n*60);
bar(X,Y,'b');
hold on

m = sum(X.*Y)/sum(Y);
var = sum(Y.*(X-m).^2)/sum(Y);
o = sqrt(var);
G = factorial(n)*(1/(sqrt(2*pi)*o))*exp(-( ((X-m).^2)/(2*(o^2)) ));

plot(X,G,'LineWidth',2);
tit = "n = " + n + " and seed = " + seed;
title(tit);
legend("Occurrences of costs","Normal distribution");
legend('boxoff')
xlabel('cost');
ylabel('occurences');
end
```

```
%% Arrays of Data

n_14 = 1:14;
n_16 = 1:16;

Brute_Force_RT = [0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.004,0.042,0.481,6.032,81.358,1189.379];
Brute_Force_MIN = [46,62,52,98,95,134,152,172,139,197,194,183,193,267];
Brute_Force_MAX = [46,72,113,167,206,249,316,333,405,472,516,568,589,658];

Branch_Bound_RT = [0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.002,0.009,0.008,0.078,3.623,3.704,20.082];
Branch_Bound_MIN = [46,62,52,98,95,134,152,172,139,197,194,183,193,267,240,262];
Branch_Bound_MAX = [46,72,113,167,206,249,316,333,405,472,516,568,589,658,680,762];

Random_RT = [0.010,0.025,0.032,0.041,0.049,0.059,0.068,0.078,0.090,0.102,0.111,0.122,0.132,0.142,0.151,0.161];
Random_MIN = [46,62,52,98,95,134,152,172,146,197,201,208,225,301,292,313];
Random_MAX = [46,72,113,167,206,249,316,333,405,465,508,539,554,609,640,701];

%% Plot Run Time
subplot(2,2,1);
plot(n_14,Brute_Force_RT,'-s');
title('Brute Force');
xlabel('n value');
ylabel('run time');

subplot(2,2,2);
plot(n_16,Branch_Bound_RT,'-ro');
title('Branch and Bound');
xlabel('n value');
ylabel('run time');

subplot(2,2,3);
plot(n_16,Random_RT,'-gx');
title('Random Permutations');
xlabel('n value');
ylabel('run time');

subplot(2,2,4);
Brute_Force_RT(15) = 1189.379;
Brute_Force_RT(16) = 1189.379;

plot(n_16,Brute_Force_RT,'-s');
hold on
plot(n_16,Branch_Bound_RT,'-ro');
plot(n_16,Random_RT,'-gx');
t="Method Comparison" + newline + "Ignore n=15,16 for BF";
title(t);
xlabel('n value');
ylabel('run time');
legend({'Brute Force','Branch and Bound','Random Permutations'},'Location','northwest');
legend('boxoff');

sgtitle('Run Time');


%% Plot Minimum Cost
subplot(2,2,1);
plot(n_14,Brute_Force_MIN,'-s');
title('Brute Force');
xlabel('n value');
ylabel('cost');

subplot(2,2,2);
plot(n_16,Branch_Bound_MIN,'-ro');
title('Branch and Bound');
xlabel('n value');
ylabel('cost');

subplot(2,2,3);
```

```
73 -    subplot(2,2,3);
74 -    plot(n_16,Random_MIN,'-gx');
75 -    title('Random Permutations');
76 -    xlabel('n value');
77 -    ylabel('cost');
78
79 -    subplot(2,2,4);
80 -    Brute_Force_MIN(15) = 267;
81 -    Brute_Force_MIN(16) = 267;
82
83 -    plot(n_16,Brute_Force_MIN,'-s');
84 -    hold on
85 -    plot(n_16,Branch_Bound_MIN,'-ro');
86 -    plot(n_16,Random_MIN,'-gx');
87 -    t="Method Comparison" + newline + "Ignore n=15,16 for BF";
88 -    title(t);
89 -    xlabel('n value');
90 -    ylabel('cost');
91 -    legend({'Brute Force','Branch and Bound','Random Permutations'},'Location','northwest');
92 -    legend('boxoff');
93
94 -    sgtitle('Min Cost');
95
96
97      %% Plot Maximum Cost
98 -    subplot(2,2,1);
99 -    plot(n_14,Brute_Force_MAX,'-s');
100 -   title('Brute Force');
101 -   xlabel('n value');
102 -   ylabel('cost');
103
104 -   subplot(2,2,2);
105 -   plot(n_16,Branch_Bound_MAX,'-ro');
106 -   title('Branch and Bound');
107 -   xlabel('n value');
108 -   ylabel('cost');
109
110 -   subplot(2,2,3);
111 -   plot(n_16,Random_MAX,'-gx');
112 -   title('Random Permutations');
113 -   xlabel('n value');
114 -   ylabel('cost');
115
116 -   subplot(2,2,4);
117 -   Brute_Force_MAX(15) = 658;
118 -   Brute_Force_MAX(16) = 658;
119
120 -   plot(n_16,Brute_Force_MAX,'-s');
121 -   hold on
122 -   plot(n_16,Branch_Bound_MAX,'-ro');
123 -   plot(n_16,Random_MAX,'-gx');
124 -   t="Method Comparison" + newline + "Ignore n=15,16 for BF";
125 -   title(t);
126 -   xlabel('n value');
127 -   ylabel('cost');
128 -   legend({'Brute Force','Branch and Bound','Random Permutations'},'Location','northwest');
129 -   legend('boxoff');
130
131 -   sgtitle('Max Cost');
132
```