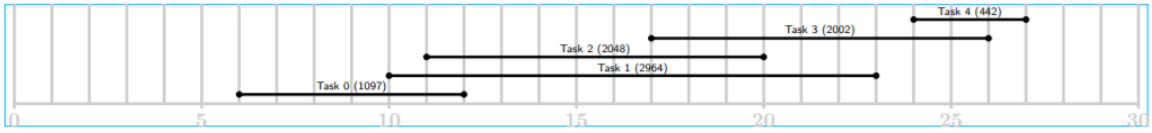
# Licenciatura em Engenharia Informática
# Algoritmos e Estruturas de Dados
# 2020/2021

## Trabalho 1 - *"Generalized weighted job selection problem"*




| task 4 | task 3 | task 2 | task 1 | task 0 | viable | total profit |
|---|---|---|---|---|---|---|
| no | no | no | no | no | yes | 0 |
| no | no | no | no | yes | yes | 1097 |
| no | no | no | yes | no | yes | 2964 |
| no | no | no | yes | yes | yes | 4061 |
| no | no | yes | no | no | yes | 2048 |
| no | no | yes | no | yes | yes | 3145 |
| no | no | yes | yes | no | yes | 5012 |
| no | no | yes | yes | yes | no | 6109 |
| no | yes | no | no | no | yes | 2002 |
| no | yes | no | no | yes | yes | 3099 |
| no | yes | no | yes | no | yes | 4966 |
| no | yes | no | yes | yes | yes | 6063 |
| no | yes | yes | no | no | yes | 4050 |
| no | yes | yes | no | yes | yes | 5147 |
| no | yes | yes | yes | no | no | 7014 |
| no | yes | yes | yes | yes | no | 8111 |

| task 4 | task 3 | task 2 | task 1 | task 0 | viable | total profit |
|---|---|---|---|---|---|---|
| yes | no | no | no | no | yes | 442 |
| yes | no | no | no | yes | yes | 1539 |
| yes | no | no | yes | no | yes | 3406 |
| yes | no | no | yes | yes | yes | 4503 |
| yes | no | yes | no | no | yes | 2490 |
| yes | no | yes | no | yes | yes | 3587 |
| yes | no | yes | yes | no | yes | 5454 |
| yes | no | yes | yes | yes | no | 6551 |
| yes | yes | no | no | no | yes | 2444 |
| yes | yes | no | no | yes | yes | 3541 |
| yes | yes | no | yes | no | yes | 5408 |
| yes | yes | no | yes | yes | yes | 6505 |
| yes | yes | yes | no | no | yes | 4492 |
| yes | yes | yes | no | yes | yes | 5589 |
| yes | yes | yes | yes | no | no | 7456 |
| yes | yes | yes | yes | yes | no | 8553 |

**André Gomes (33,3%)** **Nº 97541**
**José Trigo (33,3%)** **Nº 98597**
**Pedro Monteiro (33,3%)** **Nº 97484**

# Índice:

## Introdução:

No primeiro trabalho prático da unidade curricular de Algoritmos e Estrutura de Dados, foi proposto o problema da seleção de trabalho ponderado generalizado. Este problema consiste em determinar, dado um conjunto de X tarefas, cada uma com um prazo de começo/entrega e lucro, e Y programadores, a melhor maneira de distribuir essas tarefas entre os programadores de forma a que o lucro obtido seja maximizado, tendo em conta limitações como o facto de um programador só poder trabalhar numa tarefa de cada vez e de que não pode haver mais que um trabalhador a trabalhar na mesma tarefa.

## Abordagem do problema

Começámos por ler o enunciado, disponível na plataforma do e-learning, e analisar todo o código fornecido pelo docente, procurando saber e interpretar o que cada pedaço desse código fazia.

Percebemos então que teríamos um número de tarefas (tasks) para distribuir por um número de programadores.

Estas tasks possuem características que terão de ser verificadas para que possam ser distribuídas pelos programadores, tais como:

- starting date
- ending date
- profit

Verificamos logo que uma task pode ser feita apenas por um programador, que não pode trabalhar simultaneamente em mais do que uma tarefa.

Uma vez que o programador começa uma task terá de a terminar, ou seja, está ocupado até à sua *ending date*, o que implica não poder trabalhar noutra.

Ao longo da resolução desenvolvemos diferentes soluções para a implementação do código, começando por conseguirmos apenas realizar 17 tasks.

Após vários testes conseguimos melhorar um pouco, pelo que passámos a conseguir resolver 26 tasks.

Por fim, o programa funciona até 62 tarefas e 10 programadores, sendo que às 63 ocorre overflow, embora só tenha sido testado até 40 e no relatório seja tratado até 32.

## Código Utilizado

```
void gen_binary_combinations(int len, short *bits) {
  int counter=0;
  //ESTE FOR É CHAMADO 2^LEN -1 VEZES
  for (int i = 0; i < 1 << len; ++i)
  {
    for (int j = len - 1; j >= 0; --j) // neste for gera-se cada bit
    {
      bits[counter] = ((1<<j)&i) > 0; //((1<<j)&i) > 0 evaluates to 1 if
the jth bit is set, 0 otherwise
      counter++;
    }

}
```

Inicialmente começámos por implementar uma função que gerasse todas as combinações binárias possíveis até 2^T (T= nº de tarefas) e guardasse todos os bits gerados (2^T * T, em que 2^T representa o número de combinações e é multiplicado por T pois cada combinação tem T tarefas, obtendo assim o número total de elementos do array) num array para posteriormente serem lidos numa janela deslizante de tamanho T (estratégia semelhante à utilizada no Codificador LZ77 lecionado na disciplina de Sistemas Multimédia). Rapidamente nos apercebemos que esta abordagem seria imensamente dispendiosa em termos de memória.

De modo a solucionar o problema da eficiência ao nível da memória, recorremos ao poder da Internet para encontrar uma forma de criar um array de bits em vez de um array de ints, coisa que não existe em C, mas com uns truques bastante inteligentes explicados nesta publicação (guardando os bits dentro do int): (link na bibliografia) conseguimos alcançar o pretendido.

Corremos o programa de novo para um número elevado de tarefas e percebemos que mesmo assim a memória utilizada era excessiva e não nos permitiria chegar ao nosso objetivo: 32 tarefas.

Finalmente, apercebemo-nos de que não precisávamos de guardar as combinações num array e que podíamos trabalhar diretamente com os valores gerados, resultando neste código, que iremos explicar agora em detalhe:

```
void awesome_implementation(problem_t *problem) {
  int k, combination_profit,max_profit,tasks_done,max_n_of_tasks,
task_idx, retval;
  long long i,j,viable_job_selections;

  max_profit = 0;
  viable_job_selections = pow(2, problem->T);
  max_n_of_tasks = 0;
  FILE *out = fopen("viable_job_selections_profits.txt","w");

  for (i = 0; i < 1LL << (long long) problem->T; ++i)//ESTE FOR É
CHAMADO 2^LEN -1 VEZES
  {
    memset(problem->busy, -1, problem->P * sizeof(int)); //limpar o
array busy para cada combinação
    combination_profit = 0;
    tasks_done = 0;
    task_idx = 0;
    retval = 0;
    for (j = problem->T - 1; j >= 0; --j) //este for gera cada bit da
combinação
    {
      if (((1LL << j) & i) > 0LL) { //((1<<j)&i) > 0 evaluates to 1 if
the jth bit is set, 0 otherwise
        problem->task[task_idx].assigned_to = -1;
        for (k = 0; k < problem->P; k++)
        {
          if (problem->task[task_idx].starting_date > problem->busy[k])
          { //verificar se o programador está disponivel para trabalhar
nessa data
            problem->task[task_idx].assigned_to = k;   //atribuir a
tarefa ao P
            problem->busy[k] = problem->task[task_idx].ending_date;
//atualizar a disponibilidade do P
            combination_profit = combination_profit +
problem->task[task_idx].profit;
            tasks_done++;
            break; // terminar o ciclo for
```

```
          }
        }
        if (problem->task[task_idx].assigned_to == -1) {
          retval = 1;
        }
      }
      task_idx++;
    }
    if (combination_profit > max_profit) //se a combinação for viavel,
comparar o profit
    {
      max_profit = combination_profit;
    }
    if (tasks_done > max_n_of_tasks) {
      max_n_of_tasks = tasks_done;
    }
    if (retval) { //Se a tarefa é para ser feita e não for atribuida
      viable_job_selections--;
    } else {
      fprintf(out, "%d\n", combination_profit);
    }
  }
  problem->total_profit = max_profit;
  printf("VIABLE JOB SELECTIONS: %lld\nPROFIT: %d\nLARGEST NUMBER OF
PROGRAMMING TASKS: %d\n", viable_job_selections, problem->total_profit,
max_n_of_tasks);
  fclose(out);
}
```

O primeiro ciclo for serve para iterar cada combinação, o segundo ciclo for itera cada tarefa e a condição **((1<<j)&i) > 0** devolve 1 caso seja para fazer a tarefa e 0 caso não seja para fazer a tarefa.

```
      for (i = 0; i < 1LL << (long long) problem->T; ++i) {
            for (j = problem->T - 1; j >= 0; --j) {
                  if (((1LL << j) & i) > 0LL) { ... }
                  ...
            }
            ...
}
```

Para cada combinação, limpamos o array que contém a disponibilidade dos programadores, o número de tarefas feitas, o index da tarefa e uma variável return value que é usada para contar o número de *viable job selections.*

```
   memset(problem->busy, -1, problem->P * sizeof(int)); //limpar o array
   combination_profit = 0;
   tasks_done = 0;
   task_idx = 0;
   retval = 0;
```

De seguida, para cada tarefa na combinação, se for para fazer a tarefa (if (((1LL << j) & i) > 0LL)), executamos um ciclo for que percorre os programadores e verifica se eles estão disponíveis para lhes ser atribuída a tarefa. Caso estejam disponíveis, a tarefa é lhes atribuída, a disponibilidade do programador é atualizada no array *busy*, o profit total da combinação é atualizado (soma-se o profit da tarefa que acabou de ser atribuída ao profit anterior) e ocorre um *break* para terminar o ciclo for. Caso contrário, o ciclo for é iterado novamente e verifica-se a disponibilidade do próximo programador.

```
       for (k = 0; k < problem->P; k++)
        {
          if (problem->task[task_idx].starting_date > problem->busy[k])
          { //verificar se o programador está disponivel para trabalhar
nessa data
            problem->task[task_idx].assigned_to = k;   //atribuir a
tarefa ao P
            problem->busy[k] = problem->task[task_idx].ending_date;
//atualizar a disponibilidade do P
            combination_profit = combination_profit +
problem->task[task_idx].profit;
            tasks_done++;
            break; // terminar o ciclo for
          }
```

Se após o ciclo for, uma tarefa que devia ser atribuída não ficar atribuída (por falta de disponibilidade dos programadores), atualizamos a variável retval para 1, que é uma variável de controlo para podermos decrementar o número de *viable job selections*. Caso contrário, escrevemos o *profit* dessa combinação para um ficheiro de texto, para mais tarde construirmos um histograma da ocorrência de cada *profit* (passo opcional).

```
if (((1LL << j) & i) > 0LL) {
      for (k = 0; k < problem->P; k++) {...}
      if (problem->task[task_idx].assigned_to == -1) {
          retval = 1;
      }
}
```

(...)

```
   if (retval) { //Se a tarefa é para ser feita e não for atribuida
      viable_job_selections--;
   } else {
      fprintf(out, "%d\n", combination_profit);
   }
```

As seguintes condições comparam o profit e o número de tarefas feitas na atual iteração com o maior *profit* e o maior número de tarefas feitas até ao momento e se for superior, atualiza o valor.

```
   if (combination_profit > max_profit) //se a combinação for viavel,
comparar o profit
    {
      max_profit = combination_profit;
    }
    if (tasks_done > max_n_of_tasks) {
      max_n_of_tasks = tasks_done;
    }
```

## Resultados Obtidos

Para cada número mecanográfico obtivemos os respetivos gráficos com os tempos de execução e um gráfico que contém a variação do profit, com recurso ao matlab, através de funções tais como plot(), para fazer gráficos 2D, e surfl(), para fazer gráficos 3D.
Obtivemos também uma matriz que contém todos os tempos de execução organizados por linhas e colunas, e um histograma com o número de ocorrências de cada profit

## Nmec 97484

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0001 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0001 | 0.0000 | 0.0000 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 |
| 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 |
| 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0002 | 0.0001 |
| 0.0002 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0004 | 0.0005 | 0.0006 |
| 0.0010 | 0.0010 | 0.0010 | 0.0010 | 0.0010 | 0.0010 | 0.0010 | 0.0011 | 0.0010 | 0.0011 |
| 0.0017 | 0.0018 | 0.0020 | 0.0020 | 0.0020 | 0.0020 | 0.0030 | 0.0022 | 0.0015 | 0.0014 |
| 0.0024 | 0.0026 | 0.0026 | 0.0026 | 0.0027 | 0.0027 | 0.0026 | 0.0029 | 0.0029 | 0.0028 |
| 0.0046 | 0.0048 | 0.0052 | 0.0054 | 0.0053 | 0.0070 | 0.0056 | 0.0055 | 0.0056 | 0.0067 |
| 0.0097 | 0.0105 | 0.0110 | 0.0106 | 0.0112 | 0.0115 | 0.0110 | 0.0126 | 0.0123 | 0.0122 |
| 0.0175 | 0.0092 | 0.0092 | 0.0090 | 0.0100 | 0.0107 | 0.0103 | 0.0099 | 0.0099 | 0.0106 |
| 0.0171 | 0.0183 | 0.0149 | 0.0132 | 0.0140 | 0.0148 | 0.0149 | 0.0150 | 0.0141 | 0.0150 |
| 0.0244 | 0.0251 | 0.0255 | 0.0279 | 0.0299 | 0.0297 | 0.0301 | 0.0301 | 0.0336 | 0.0319 |
| 0.0562 | 0.0515 | 0.0566 | 0.0575 | 0.0600 | 0.0621 | 0.0624 | 0.0651 | 0.0612 | 0.0602 |
| 0.0989 | 0.1056 | 0.1147 | 0.1190 | 0.1184 | 0.1323 | 0.1328 | 0.1232 | 0.1340 | 0.1305 |
| 0.2039 | 0.2139 | 0.2186 | 0.2413 | 0.2468 | 0.2501 | 0.2504 | 0.2677 | 0.2514 | 0.2754 |
| 0.4081 | 0.4413 | 0.4519 | 0.5039 | 0.4989 | 0.5056 | 0.5083 | 0.5516 | 0.5217 | 0.5795 |
| 0.8341 | 0.8832 | 0.9415 | 1.0150 | 1.0540 | 1.0320 | 1.0240 | 1.1360 | 1.1510 | 1.1180 |
| 1.7150 | 1.8670 | 2.0220 | 1.9420 | 2.0390 | 2.0650 | 2.1210 | 2.2520 | 2.3200 | 2.3030 |
| 3.3830 | 3.8370 | 3.9520 | 4.1500 | 4.3100 | 4.4220 | 4.3850 | 4.7660 | 4.8260 | 4.7250 |
| 7.0570 | 7.8270 | 8.2630 | 8.6050 | 8.3860 | 8.7900 | 8.8180 | 9.7560 | 9.4940 | 9.5560 |
| 14.5000 | 15.4000 | 16.6600 | 16.5500 | 17.4100 | 18.4400 | 18.9100 | 18.3600 | 18.8300 | 19.9300 |
| 30.1400 | 32.8600 | 34.4400 | 35.0100 | 35.8700 | 36.8800 | 38.8300 | 38.5200 | 39.0700 | 40.7800 |
| 59.6400 | 62.8900 | 69.4300 | 71.5900 | 71.7800 | 78.1800 | 77.2100 | 77.5200 | 79.4700 | 82.8200 |
| 123.1000 | 132.7000 | 142.6000 | 144.0000 | 150.4000 | 157.1000 | 160.0000 | 156.3000 | 164.8000 | 176.0000 |
| 246.4000 | 270.3000 | 282.8000 | 300.2000 | 308.0000 | 310.0000 | 320.6000 | 335.7000 | 333.7000 | 339.2000 |

Figura 1 - Matriz com os tempos de execução referentes ao Nmec 97484

Na figura 1 podemos observar todos os tempos de execução que correspondem ao número mecanográfico 97484, onde cada linha representa uma task e cada coluna representa um programador, por exemplo na linha 31 e coluna 1 temos o valor 123.1000, em segundos, que corresponde ao tempo de execução com 24 tasks e 1 programador.

Teremos então 32 linhas, correspondentes a 32 tarefas e 10 colunas correspondentes a 10 programadores.

Os primeiros valores são semelhantes devido às aproximações do matlab, visto serem números bastante pequenos.

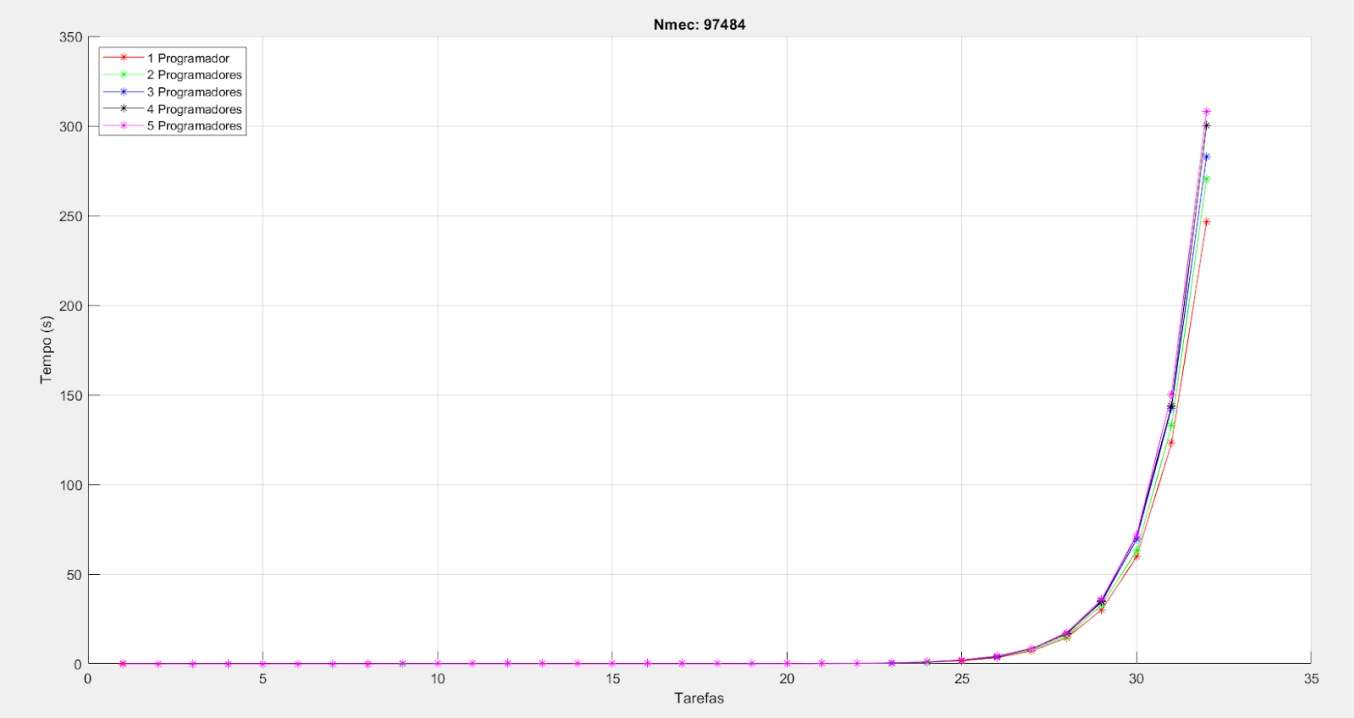



Figura 2 - Gráfico com 1 a 5 programadores

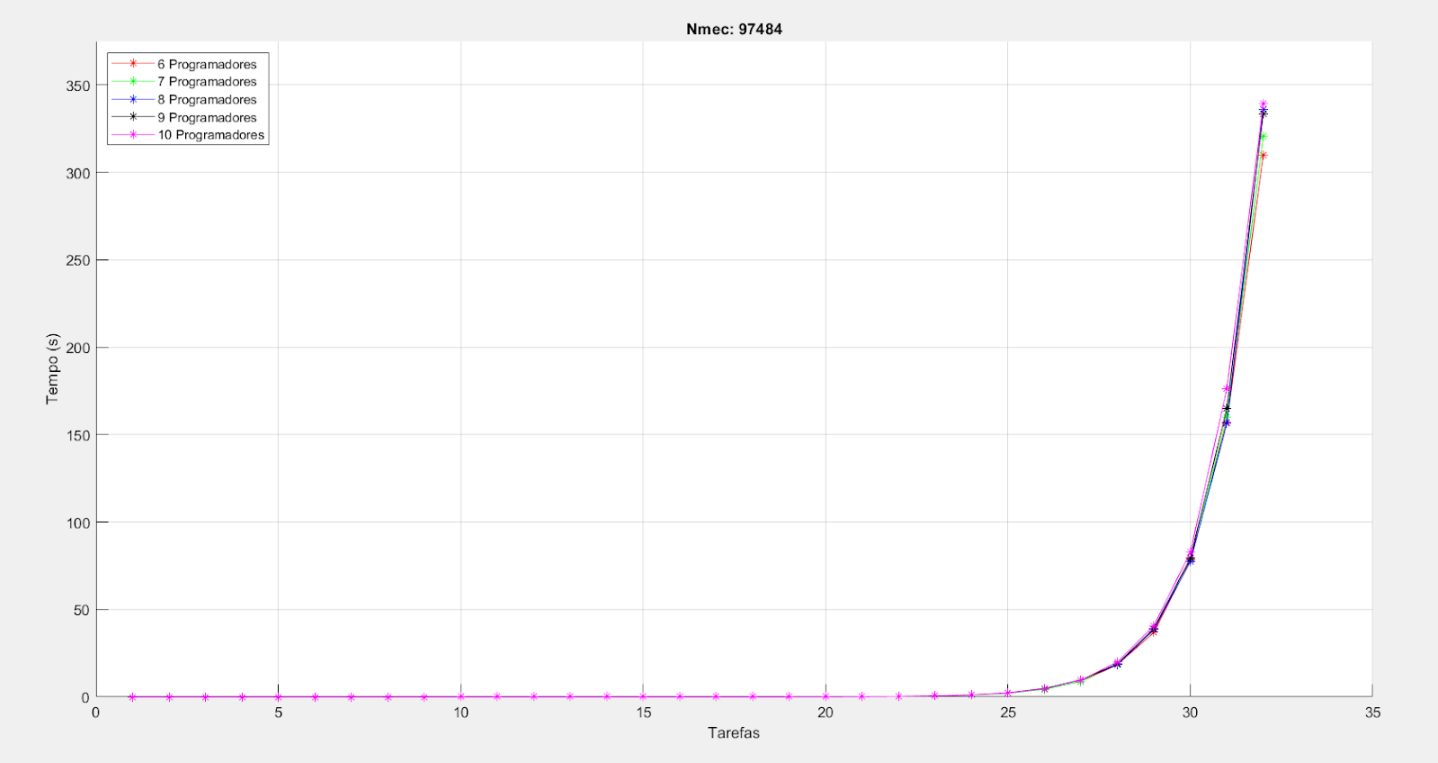



Figura 3 - Gráfico com 6 a 10 programadores

Nas figuras 2 e 3 estão representados os gráficos dos tempos de execução, em que o eixo das coordenadas corresponde ao número de tarefas. Cada linha, com a sua respectiva cor, simboliza o número de programadores, como podemos observar na legenda das figuras.

Verificamos que com o aumento do número de tasks o tempo de execução, em segundos, aumenta também, de forma exponencial.

Podemos, então, dizer que a complexidade computacional deste programa será aproximadamente $O(n^2)$.

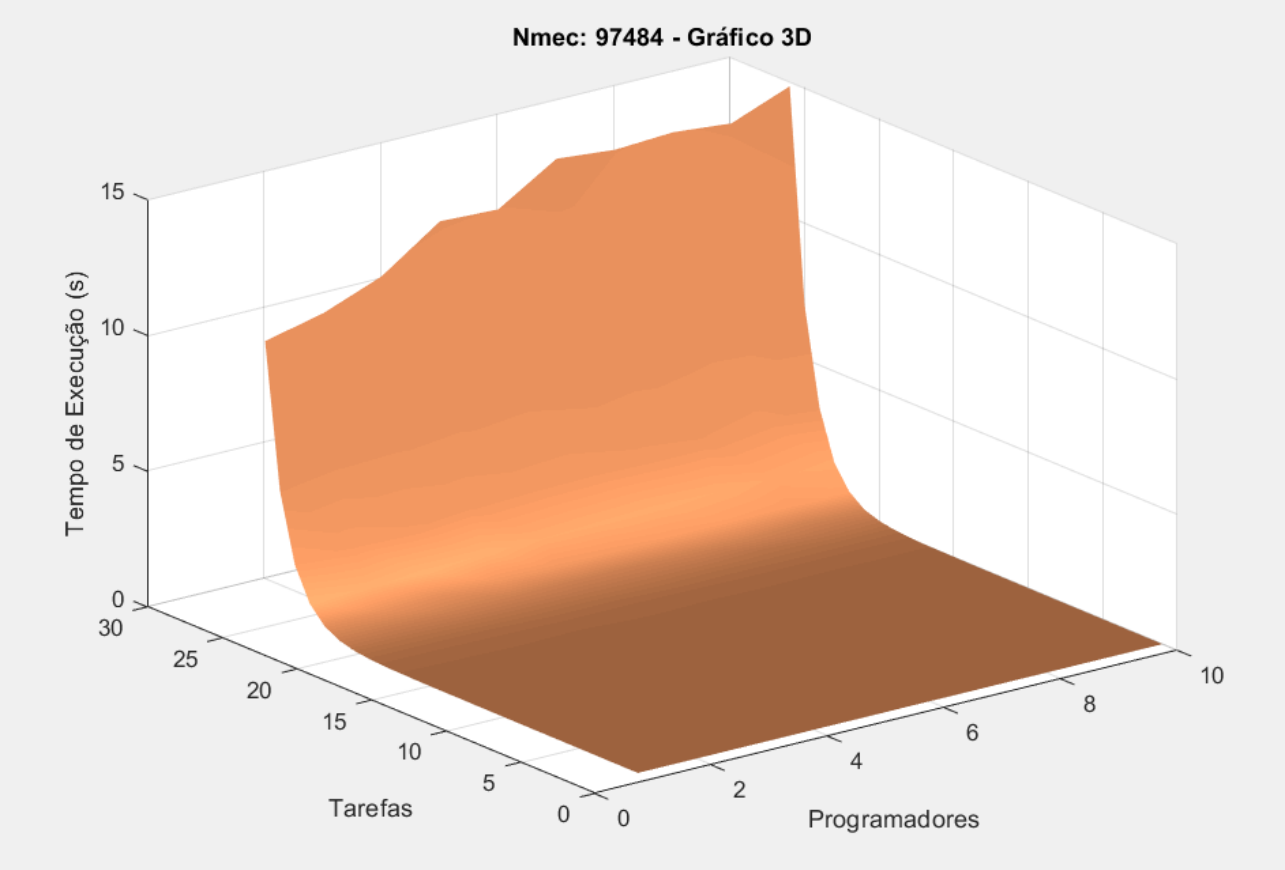



Figura 4 - Gráfico tridimensional com os tempos de execução referente ao Nmec 97484

Obtivemos ainda um gráfico tridimensional (através do uso da função surfl()), para uma melhor observação dos resultados obtidos, em que o eixo das cotas representa o tempo, o eixo das coordenadas representa os programadores e o eixo das ordenadas representa as tarefas.

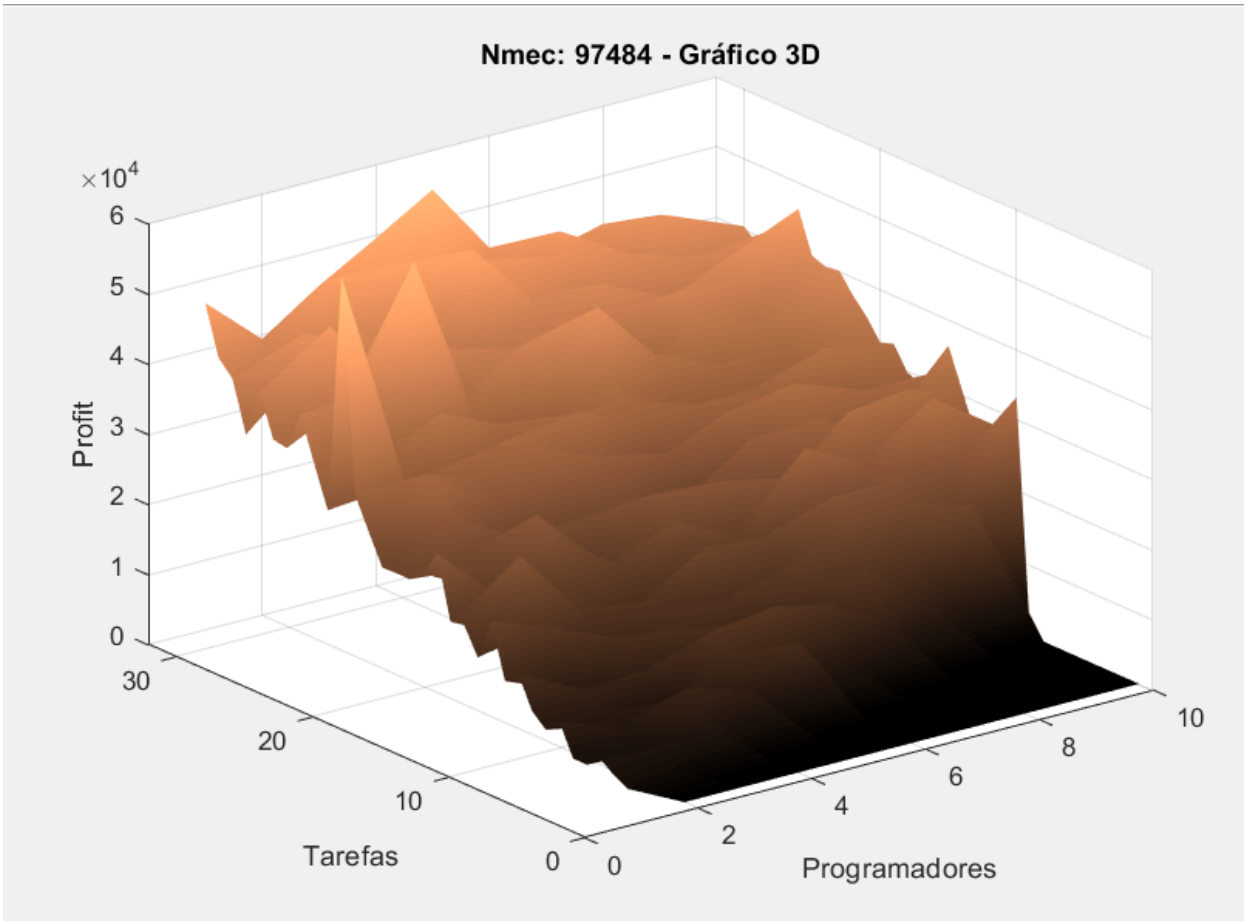



Figura 5 - Gráfico tridimensional com os valores do profit referente ao Nmec 97484

Na figura 5, podemos observar o modo como varia o valor do profit (lucro) com o número de tarefas e de programadores.

Verificamos também que o profit atinge valores máximos na ordem de $10^4$ (cerca de $6x10^4$).

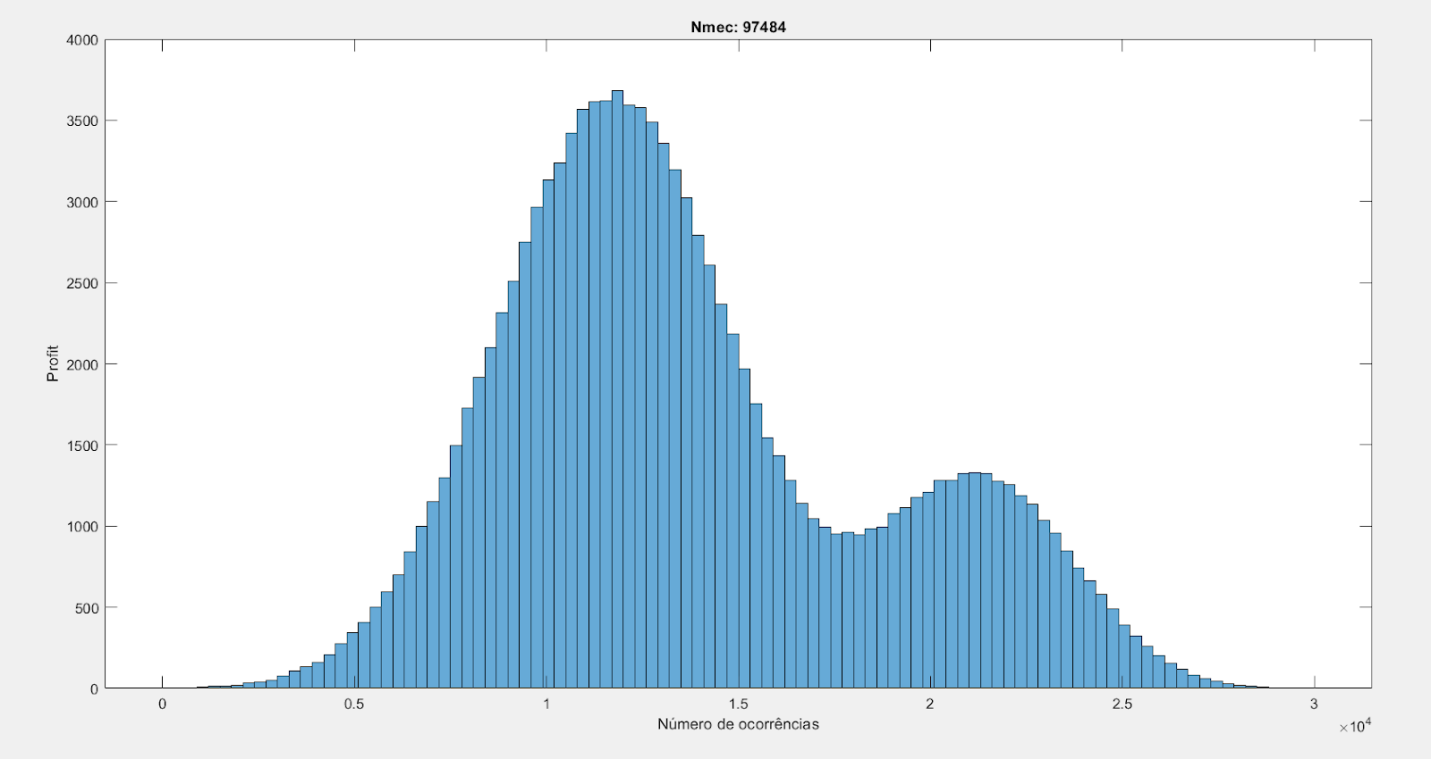



Figura 6 - Histograma com o número de ocorrências de cada profit

Através da função histogram() do matlab construímos o histograma representado na figura 6, onde procuramos mostrar o número de ocorrências de cada valor de profit.

Podemos ver, então, que, por exemplo, para um valor de lucro de 500, existem cerca de $5.5 \times 10^4$ ocorrências.

## Nmec 97541

De forma semelhante ao anterior corremos agora o programa para outro número mecanográfico, onde obtivemos os seguintes resultados:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 | 0 | 0 |
| 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 | 0 |
| 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0 |
| 0.0003 | 0.0003 | 0.0004 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 |
| 0.0005 | 0.0005 | 0.0005 | 0.0005 | 0.0005 | 0.0005 | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| 0.0002 | 0.0002 | 0.0003 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 |
| 0.0004 | 0.0004 | 0.0004 | 0.0005 | 0.0005 | 0.0005 | 0.0005 | 0.0006 | 0.0005 | 0.0006 |
| 0.0011 | 0.0012 | 0.0013 | 0.0014 | 0.0015 | 0.0014 | 0.0015 | 0.0015 | 0.0016 | 0.0019 |
| 0.0031 | 0.0030 | 0.0032 | 0.0034 | 0.0035 | 0.0036 | 0.0037 | 0.0035 | 0.0035 | 0.0037 |
| 0.0059 | 0.0061 | 0.0069 | 0.0068 | 0.0071 | 0.0072 | 0.0073 | 0.0069 | 0.0071 | 0.0074 |
| 0.0120 | 0.0126 | 0.0134 | 0.0140 | 0.0143 | 0.0149 | 0.0159 | 0.0140 | 0.0139 | 0.0090 |
| 0.0121 | 0.0129 | 0.0128 | 0.0136 | 0.0136 | 0.0146 | 0.0146 | 0.0153 | 0.0152 | 0.0153 |
| 0.0246 | 0.0263 | 0.0275 | 0.0278 | 0.0301 | 0.0293 | 0.0316 | 0.0292 | 0.0349 | 0.0332 |
| 0.0494 | 0.0528 | 0.0539 | 0.0576 | 0.0582 | 0.0622 | 0.0649 | 0.0618 | 0.0630 | 0.0644 |
| 0.1009 | 0.1060 | 0.1139 | 0.1259 | 0.1192 | 0.1253 | 0.1259 | 0.1318 | 0.1357 | 0.1300 |
| 0.2052 | 0.2164 | 0.2322 | 0.2472 | 0.2463 | 0.2603 | 0.2804 | 0.2702 | 0.2807 | 0.2827 |
| 0.4362 | 0.4686 | 0.4785 | 0.5088 | 0.5110 | 0.5305 | 0.5248 | 0.5635 | 0.5285 | 0.5923 |
| 0.8527 | 0.9188 | 0.9634 | 0.9723 | 1.0060 | 1.0440 | 1.0580 | 1.1240 | 1.1050 | 1.1450 |
| 1.7290 | 1.8350 | 1.8700 | 2.0430 | 2.0450 | 2.1440 | 2.1900 | 2.1450 | 2.2280 | 2.2460 |
| 3.5060 | 3.6070 | 3.9340 | 4.2330 | 4.2820 | 4.3320 | 4.4830 | 4.4050 | 4.5470 | 4.8930 |
| 7.0430 | 7.7480 | 8.4480 | 8.2070 | 8.7920 | 8.8600 | 8.5900 | 9.1970 | 9.6410 | 9.7600 |
| 14.5600 | 16.0800 | 16.4900 | 17.0800 | 18.1600 | 17.5700 | 18.2600 | 18.5300 | 18.5800 | 19.0500 |
| 30.2700 | 32.5600 | 33.9900 | 36.1000 | 36.8700 | 38.5300 | 37.3100 | 38.8200 | 40.0900 | 42.1100 |
| 60.8000 | 63.9600 | 68.8500 | 69.0400 | 72.6500 | 75.6700 | 76.6000 | 78.7600 | 79.3200 | 85.7100 |
| 123.5000 | 128.2000 | 142.7000 | 150.5000 | 150.9000 | 151.8000 | 154.2000 | 154.4000 | 164.2000 | 170.6000 |
| 255.9000 | 270.8000 | 288.7000 | 304.3000 | 315.4000 | 312.5000 | 329.4000 | 322.8000 | 343.0000 | 346.7000 |

Figura 7 - Matriz com os tempos de execução referentes ao Nmec 97541

Apesar de os valores iniciais da matriz serem bastante semelhantes e quase nulos, devido às aproximações do matlab, visto serem números de ordem $10^{-4}$, observamos um pequeno aumento dos tempos de execução, o que não é de todo relevante para o nosso problema.

Estas pequenas diferenças entre valores devem-se à maneira como o computador processa as combinações que foram geradas, aleatoriamente, podendo necessitar estas de mais recursos do computador.

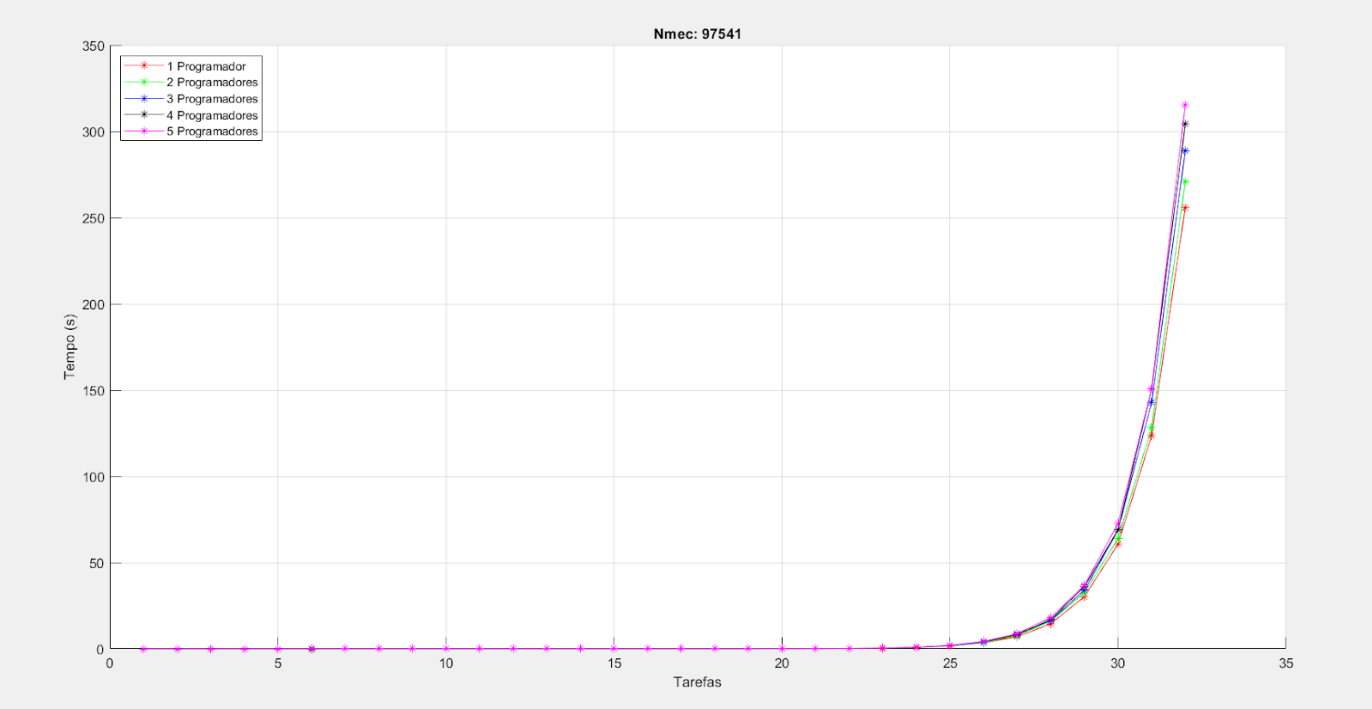



Figura 8 - Gráfico com 1 a 5 programadores

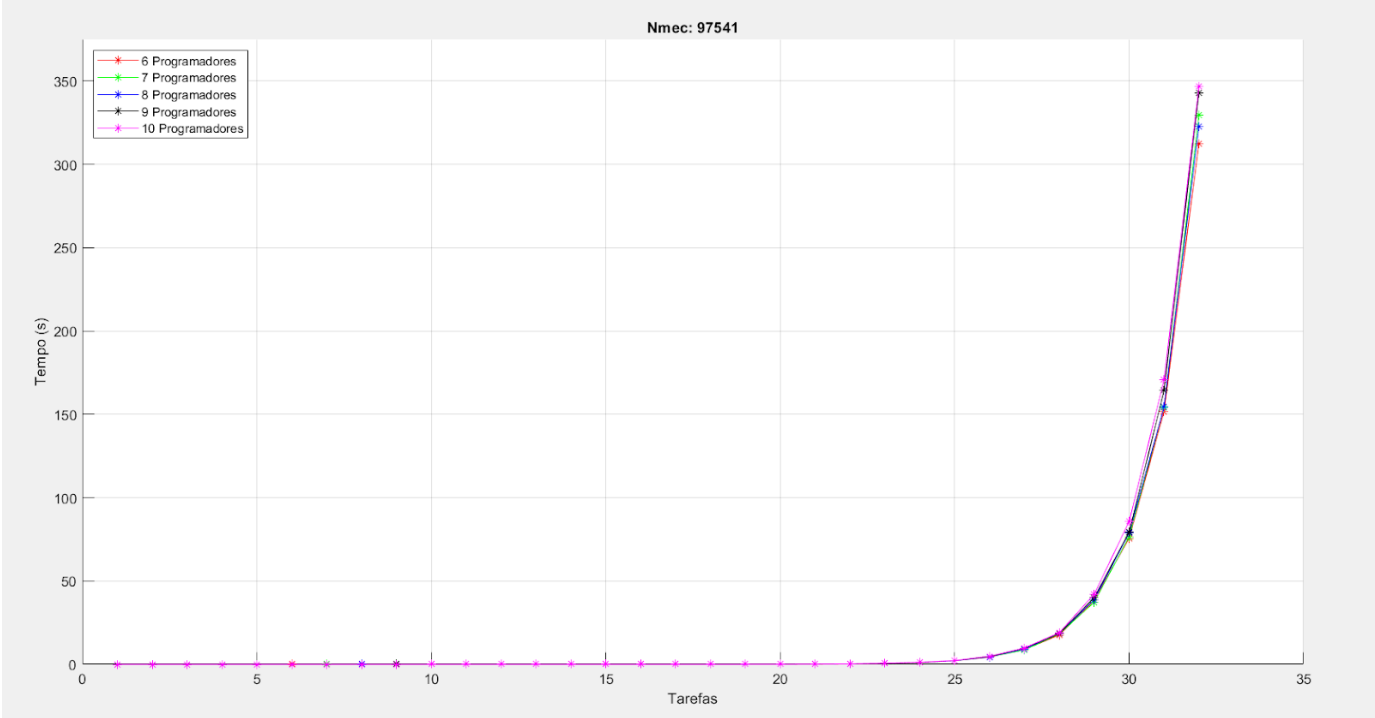



Figura 9 - Gráfico com 6 a 10 programadores

A partir dos gráficos obtidos com este número mecanográfico, e comparando com os resultados obtidos para o número mecanográfico anterior verificamos um aumento dos tempos de execução, devido ao facto de as combinações geradas, de forma aleatória, demorarem mais a ser processadas.

De igual modo, o eixo das coordenadas representa o número de tarefas e o eixo das ordenadas o tempo de execução em segundos.

Através da legenda podemos ver o número de programadores, associado a cada linha do gráfico.

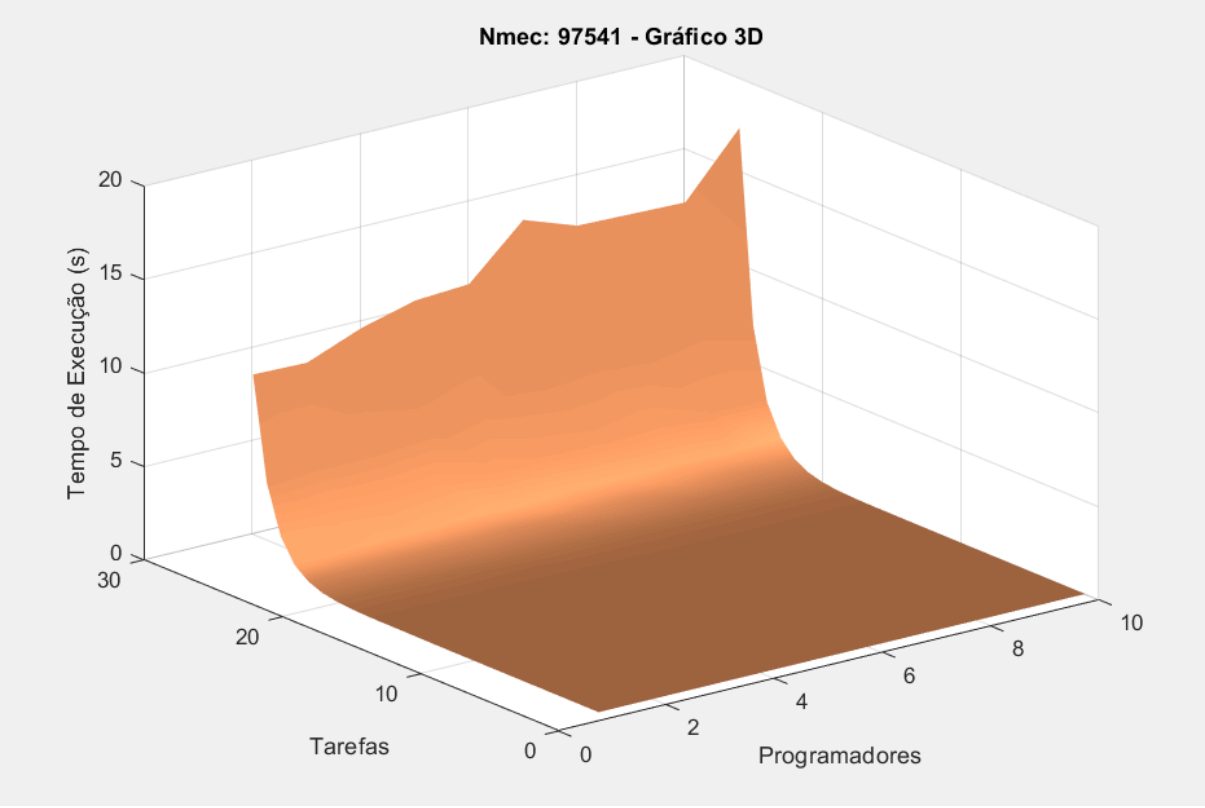



Figura 10 - Gráfico tridimensional com os tempos de execução referente ao Nmec 97541

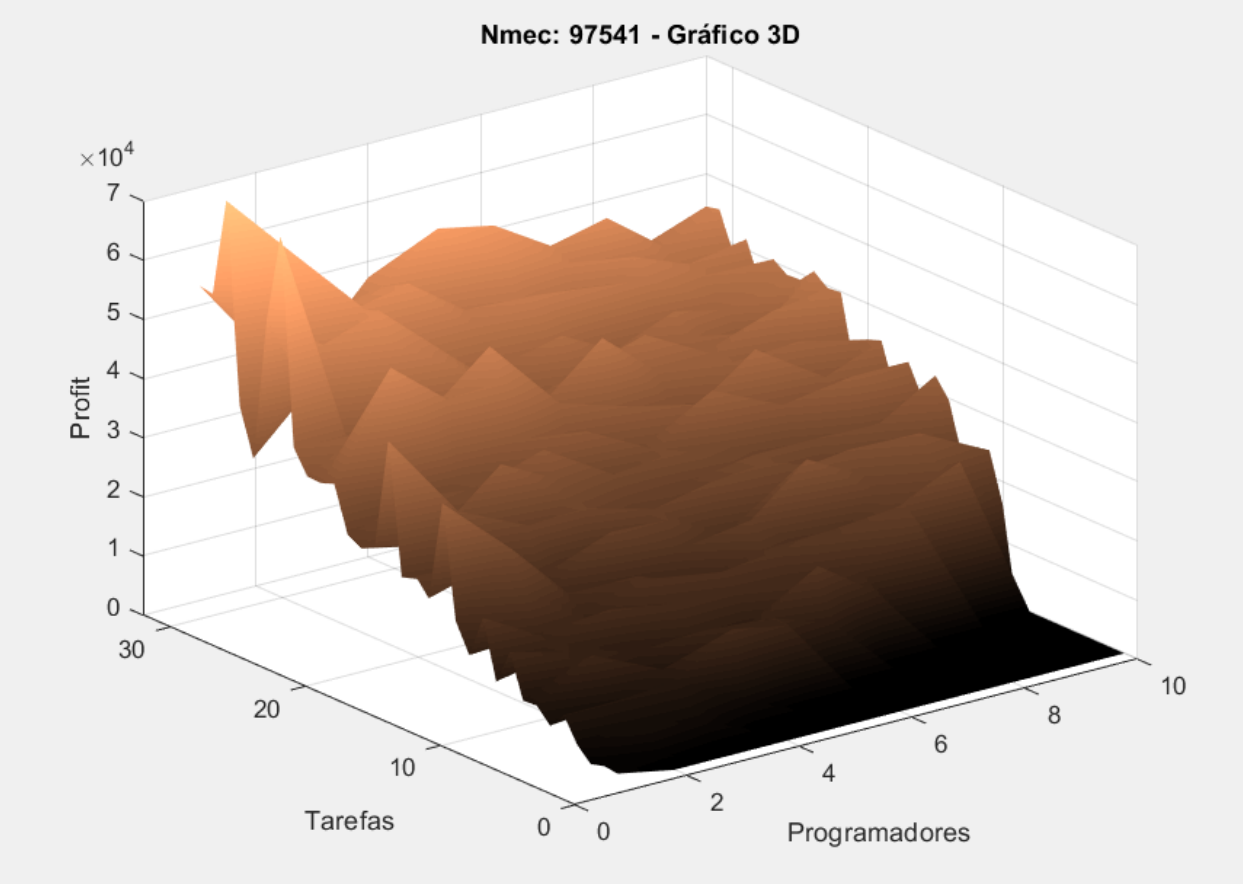



Figura 11 - Gráfico tridimensional com os valores do profit referente ao Nmec 97541

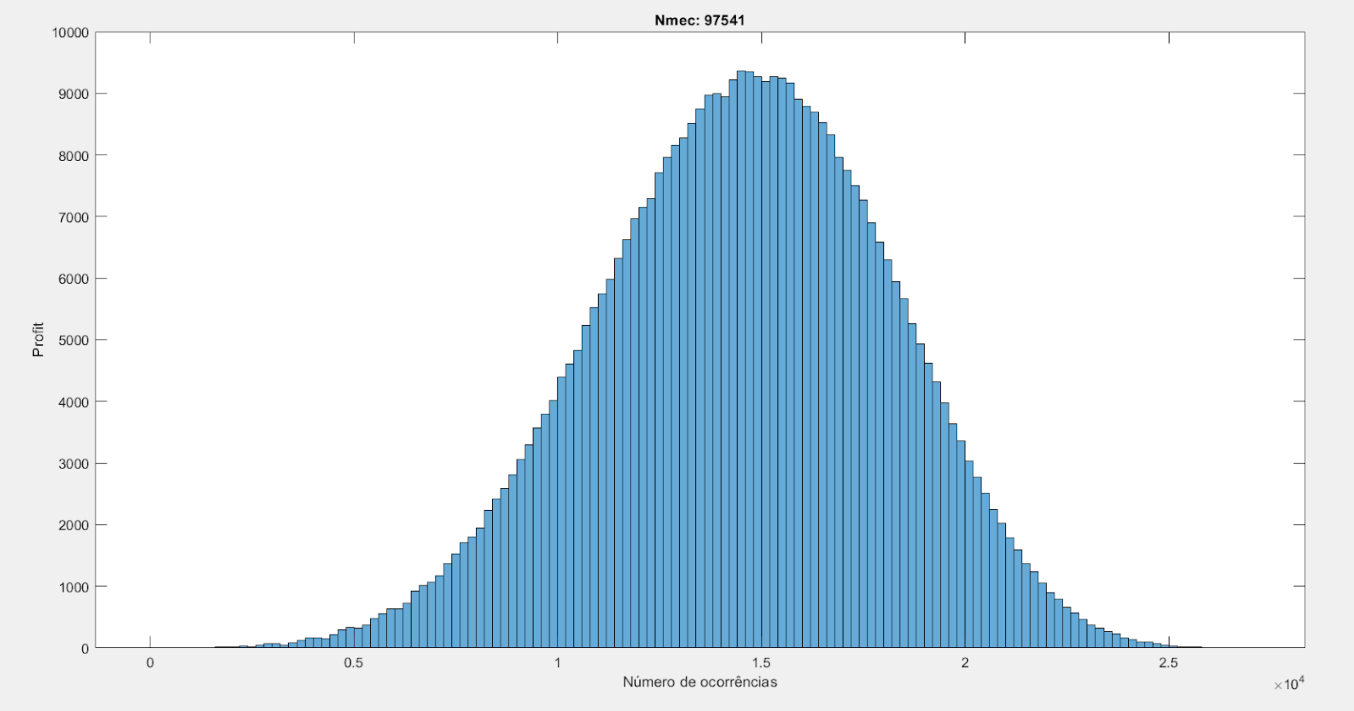



Figura 12 - Histograma com o número de ocorrências de cada profit

Com o histograma obtido na figura 12, e comparando com o histograma do primeiro número mecanográfico verificamos um aumento dos valores associados ao profit, e um aumento também do número geral de ocorrências.

## Nmec 98597

Como realizado para os dois números mecanográficos anteriores corremos também o programa de modo a obter os resultados para este número mecanográfico.

De forma semelhante obtivemos a matriz que contém todos os tempos de execução quando corrido o programa com este número mecanográfico, os gráficos contendo os tempos de execução em função das tarefas e os gráficos tridimensionais com os tempos de execução e com a variação do profit.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.0001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0001 | 0.0001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0001 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0001 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0001 | 0.0000 | 0 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 | 0 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0 |
| 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| 0.0001 | 0.0001 | 0.0001 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0001 | 0.0001 |
| 0.0002 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 | 0.0003 |
| 0.0005 | 0.0005 | 0.0005 | 0.0006 | 0.0006 | 0.0008 | 0.0008 | 0.0008 | 0.0008 | 0.0008 |
| 0.0014 | 0.0018 | 0.0024 | 0.0031 | 0.0032 | 0.0033 | 0.0033 | 0.0040 | 0.0043 | 0.0038 |
| 0.0069 | 0.0082 | 0.0078 | 0.0080 | 0.0046 | 0.0033 | 0.0036 | 0.0033 | 0.0035 | 0.0034 |
| 0.0056 | 0.0061 | 0.0063 | 0.0065 | 0.0068 | 0.0070 | 0.0068 | 0.0073 | 0.0071 | 0.0075 |
| 0.0114 | 0.0118 | 0.0128 | 0.0129 | 0.0134 | 0.0134 | 0.0138 | 0.0144 | 0.0141 | 0.0142 |
| 0.0232 | 0.0209 | 0.0138 | 0.0153 | 0.0155 | 0.0161 | 0.0162 | 0.0171 | 0.0171 | 0.0165 |
| 0.0263 | 0.0257 | 0.0279 | 0.0278 | 0.0290 | 0.0294 | 0.0302 | 0.0316 | 0.0310 | 0.0323 |
| 0.0490 | 0.0536 | 0.0533 | 0.0559 | 0.0585 | 0.0597 | 0.0618 | 0.0629 | 0.0660 | 0.0599 |
| 0.1013 | 0.1047 | 0.1070 | 0.1152 | 0.1186 | 0.1236 | 0.1308 | 0.1283 | 0.1338 | 0.1257 |
| 0.2000 | 0.2075 | 0.2236 | 0.2355 | 0.2376 | 0.2443 | 0.2642 | 0.2762 | 0.2599 | 0.2657 |
| 0.4064 | 0.4362 | 0.4552 | 0.4869 | 0.5085 | 0.5181 | 0.5262 | 0.5427 | 0.5349 | 0.5656 |
| 0.8443 | 0.9186 | 0.9356 | 0.9904 | 0.9779 | 1.0750 | 1.1020 | 1.1130 | 1.0850 | 1.1960 |
| 1.7410 | 1.8700 | 1.9330 | 2.1080 | 2.1400 | 2.2470 | 2.2310 | 2.1680 | 2.2320 | 2.4210 |
| 3.5410 | 3.7320 | 3.8030 | 4.0850 | 4.2760 | 4.3510 | 4.4090 | 4.5010 | 4.5810 | 4.6920 |
| 6.9660 | 7.7410 | 7.8290 | 8.5220 | 8.7110 | 9.1030 | 8.9630 | 8.9720 | 9.3750 | 9.7040 |
| 14.6100 | 15.2800 | 16.2000 | 17.9300 | 17.9100 | 17.3900 | 18.8200 | 19.0500 | 20.2900 | 18.7400 |
| 29.6400 | 31.5500 | 34.2100 | 34.9600 | 37.6100 | 36.3400 | 37.8700 | 38.2100 | 40.9900 | 42.3300 |
| 61.6700 | 64.4800 | 66.7900 | 73.4200 | 74.4600 | 75.4500 | 72.3100 | 79.1500 | 79.0700 | 82.0000 |
| 123.5000 | 130.2000 | 141.1000 | 142.7000 | 153.0000 | 153.2000 | 153.2000 | 161.3000 | 163.2000 | 163.8000 |
| 257.6000 | 271.1000 | 281.5000 | 296.1000 | 303.4000 | 319.7000 | 314.5000 | 326.3000 | 337.2000 | 334.6000 |

Figura 13 - Matriz com os tempos de execução referentes ao Nmec 98597

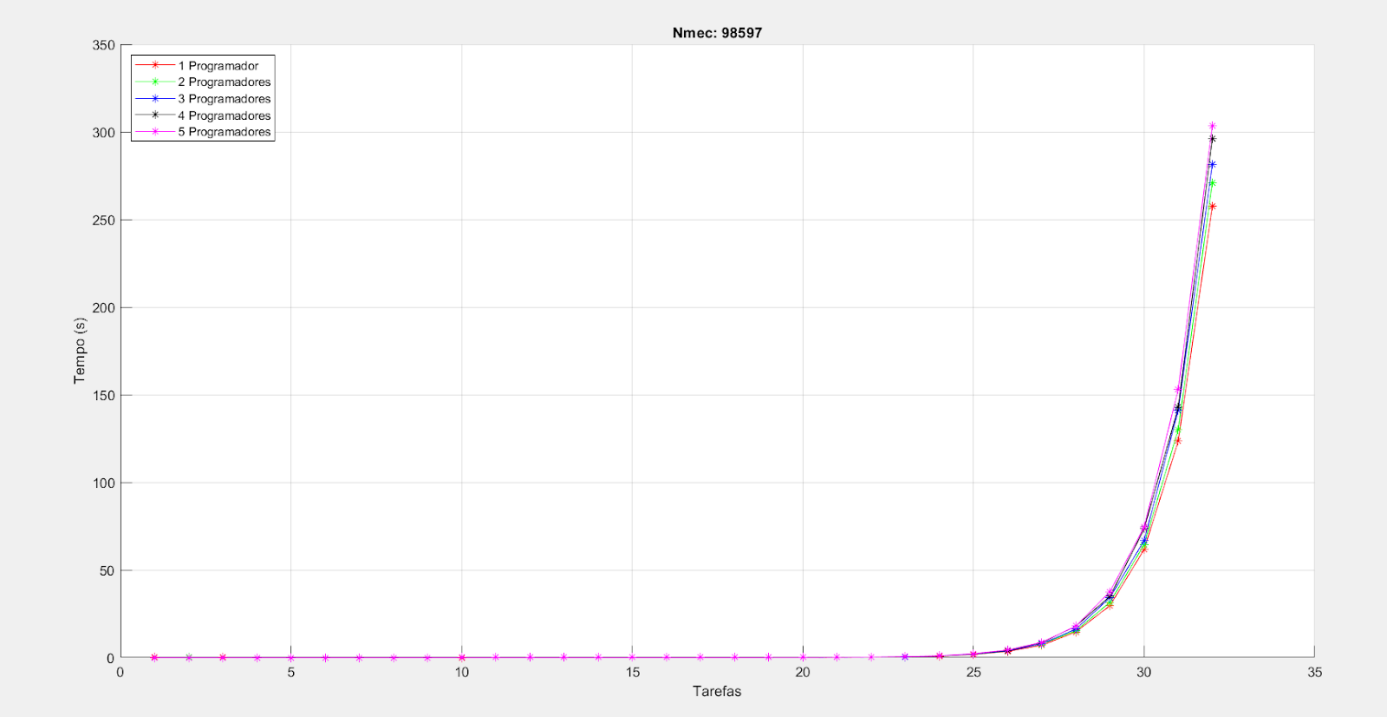



Figura 14 - Gráfico com 1 a 5 programadores

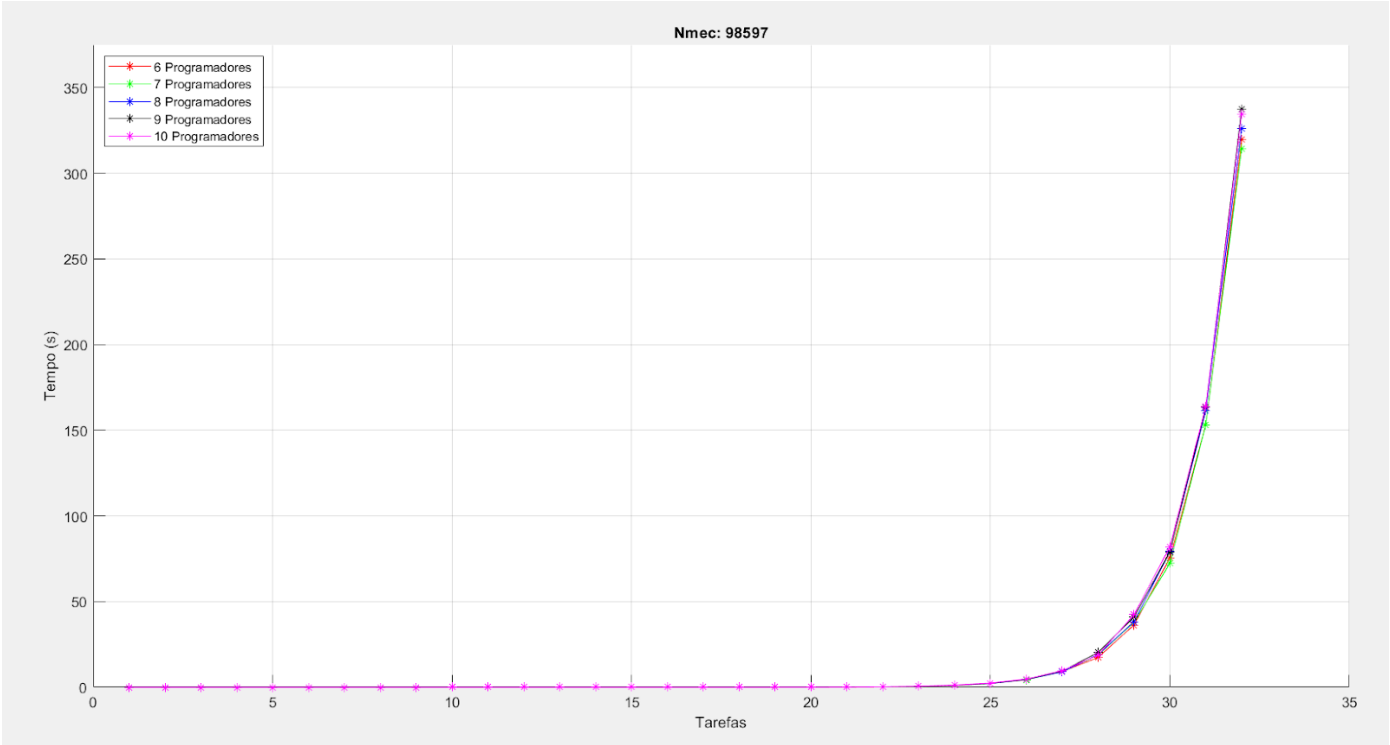



Figura 15 - Gráfico com 6 a 10 programadores

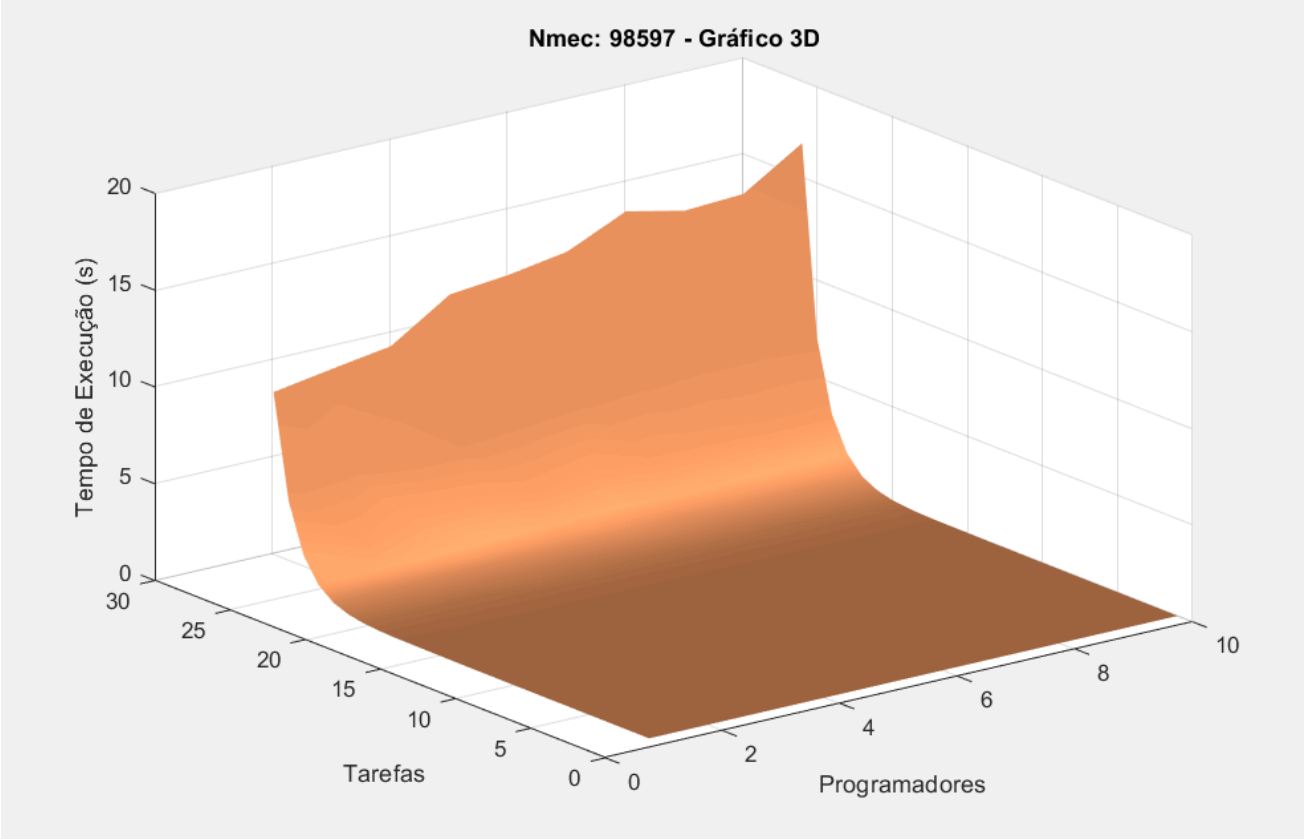



Figura 16 - Gráfico tridimensional referente ao Nmec 98597

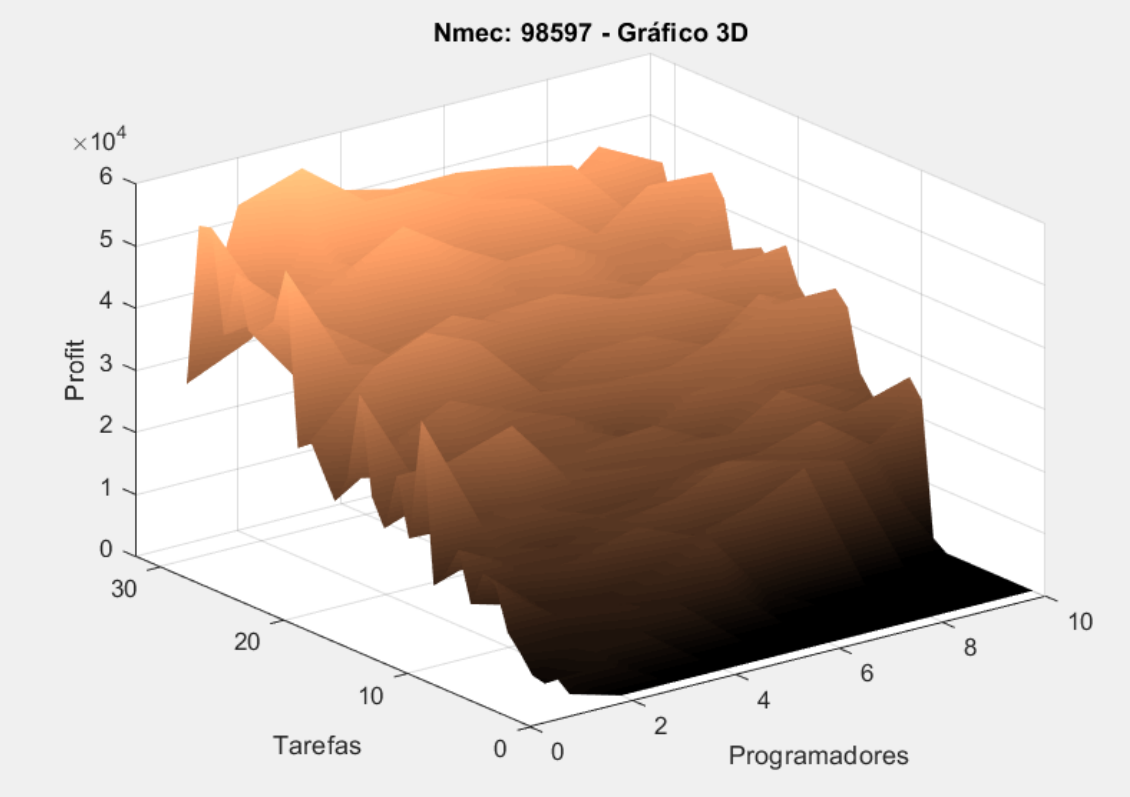



Figura 17 - Gráfico tridimensional com os valores de profit referente ao Nmec 98597

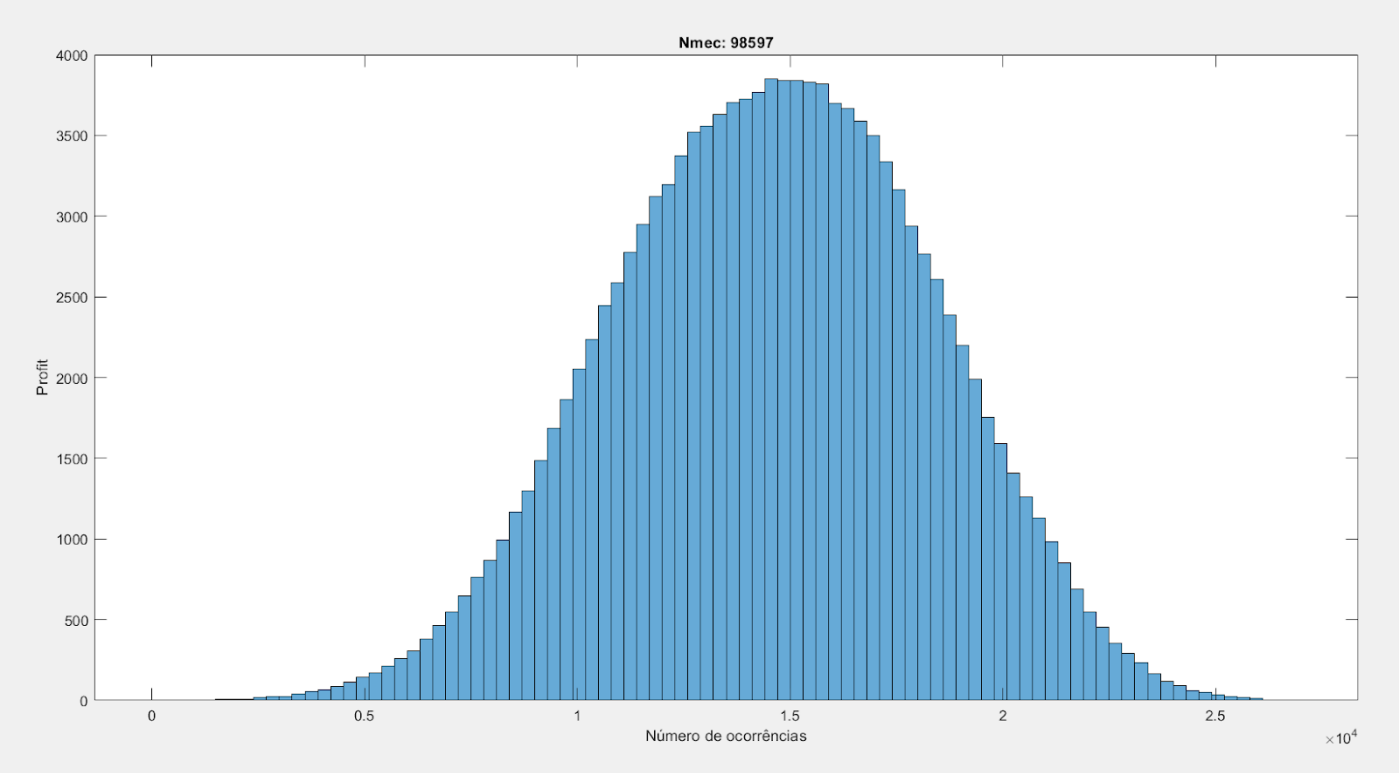



Figura 18 - Histograma com o número de ocorrências de cada profit

Em relação a todos resultados obtidos podemos observar que há ligeiras diferenças a nível dos tempos de execução, do número de tasks viáveis e dos valores do profit, para os diferentes números mecanográficos.

Estas diferenças devem-se à geração das tarefas, que são aleatórias, tendo como seed o número mecanográfico. Sendo estas tarefas geradas aleatoriamente, podem ser mais simples ou mais difíceis de processar, pelo que é normal haver pequenas diferenças entre os tempos de execução. O tempo pode também ser afetado pelos componentes do computador utilizado, visto que um computador com um melhor processador consegue processar e gerar a informação de um modo mais rápido.

## Conclusão

No desenvolvimento deste trabalho, apercebemo-nos de variados acontecimentos relacionados com conteúdos apresentados na parte teórica desta unidade curricular.

Apercebemo-nos, no que toca ao tema da complexidade computacional, do quão importante é como e onde definimos as variáveis, sendo a diferença entre definir um “int” e um “short” a mesma que gastar megas ou gigas de RAM. Tiramos também uma conclusão semelhante no que toca ao local onde declaramos as variáveis, sendo muito mais dispendioso declará-las dentro de um ciclo ou invés de declará-las anteriormente, e, dentro do ciclo, apenas alterar o seu valor.

Além disto, foi também interessante a análise do problema, pois, mesmo tratando-se de uma “situação hipotética”, este problema poderia muito bem ser um trabalho de programação no mundo real, podendo assim concluir quanto à importância que está presente no desenvolvimento de um código meticuloso e não ambíguo.

# Bibliografia

https://stackoverflow.com/questions/52014292/print-all-binary-numbers-of-length-n
https://stackoverflow.com/questions/2525310/how-to-define-and-work-with-an-array-of-bits-in-c
https://octave.sourceforge.io/octave/function/surfl.html

## Apêndice de todo o código

```
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//
// AED, 2020/2021
//
// TODO: 98597 Jose Trigo
// TODO: 97484 Pedro Monteiro
// TODO: 97541 Andre Gomes
//
// Brute-force solution of the generalized weighted job selection
problem
//
// Compile with "cc -Wall -O2 job_selection.c -lm" or equivalent
//
// In the generalized weighted job selection problem we will solve here
we have T programming tasks and P programmers.
// Each programming task has a starting date (an integer), an ending
date (another integer), and a profit (yet another
// integer). Each programming task can be either left undone or it can
be done by a single programmer. At any given
// date each programmer can be either idle or it can be working on a
programming task. The goal is to select the
// programming tasks that generate the largest profit.
//
// 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)
//      Solve the problem for each student number of the group and for
//        N=1, 2, ..., as higher as you can get and
//        P=1, 2, ... min(8,N)
//      Present the best profits in a table (one table per student
number).
//      Present all execution times in a graph (use a different color
for the times of each student number).
//      Draw the solutions for the highest N you were able to do.
//   3. (optional)
//      Ignore the profits (or, what is the same, make all profits
equal); what is the largest number of programming
```

```
//      tasks that can be done?
//   4. (optional)
//      Count the number of valid task assignments. Calculate and
display an histogram of the number of occurrences of
//      each total profit. Does it follow approximately a normal
distribution?
//   5. (optional)
//      Try to improve the execution time of the program (use the
branch-and-bound technique).
//      Can you use divide and conquer to solve this problem?
//      Can you use dynamic programming to solve this problem?
//   6. (optional)
//      For each problem size, and each student number of the group,
generate one million (or more!) valid random
//      assignments and compute the best solution found in this way.
Compare these solutions with the ones found in
//      item 2.
//   7. (optional)
//      Surprise us, by doing something more!
//   8. (mandatory)
//      Write a report explaining what you did. Do not forget to put all
your code in an appendix.
//

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "elapsed_time.h"
#include <unistd.h> /* sysconf(3) */

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//
// Random number generator interface (do not change anything in this
code section)
//
// In order to ensure reproducible results on Windows and GNU/Linux, we
use a good random number generator, available at
//   https://www-cs-faculty.stanford.edu/~knuth/programs/rng.c
// This file has to be used without any modifications, so we take care
of the main function that is there by applying
```

```
// some C preprocessor tricks
//

#define main  rng_main                          // main gets replaced by
rng_main
#ifdef __GNUC__
int rng_main() __attribute__((__unused__));   // gcc will not complain
if rnd_main() is not used
#endif
#include "rng.c"
#undef main                                     // main becomes main again

#define srandom(seed)  ran_start((long)seed)  // start the pseudo-random
number generator
#define random()       ran_arr_next()         // get the next
pseudo-random number (0 to 2^30-1)


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//
// problem data (if necessary, add new data fields in the structures; do
not change anything else in this code section)
//
// on the data structures declared below, a comment starting with
// * a I means that the corresponding field is initialized by
init_problem()
// * a S means that the corresponding field should be used when trying
all possible cases
// * IS means both (part initialized, part used)
//

#if 1

#define MAX_T  64  // maximum number of programming tasks
#define MAX_P  10  // maximum number of programmers

typedef struct
{
  int starting_date;      // I starting date of this task
  int ending_date;        // I ending date of this task
  int profit;             // I the profit if this task is performed
  int assigned_to;        // S current programmer number this task is
assigned to (use -1 for no assignment)
```

```
}
task_t;

typedef struct
{
  int NMec;               // I  student number
  int T;                  // I  number of tasks
  int P;                  // I  number of programmers
  int I;                  // I  if 1, ignore profits
  int total_profit;       // S  current total profit
  double cpu_time;        // S  time it took to find the solution
  task_t task[MAX_T];     // IS task data
  int busy[MAX_P];         // S  for each programmer, record until when
she/he is busy (-1 means idle)
  char dir_name[16];      // I  directory name where the solution file
will be created
  char file_name[64];     // I  file name where the solution data will
be stored
}
problem_t;

//Esta função compara os starting dates de duas tasks e 1 para o qsort
se a task1 começar mais cedo
//e retorna -1 ao qsort (reverse order?) se começarem ao mesmo tempo e
acabar mais cedo
int compare_tasks(const void *t1,const void *t2)
{
  int d1,d2;

  d1 = ((task_t *)t1)->starting_date;
  d2 = ((task_t *)t2)->starting_date;
  if(d1 != d2)
    return (d1 < d2) ? -1 : +1;
  d1 = ((task_t *)t1)->ending_date;
  d2 = ((task_t *)t2)->ending_date;
  if(d1 != d2)
    return (d1 < d2) ? -1 : +1;
  return 0;
}

void init_problem(int NMec,int T,int P,int ignore_profit,problem_t
*problem)
{
  int i,r,scale,span,total_span;
```

```
  int *weight;

  //
  // input validation
  //
  if(NMec < 1 || NMec > 999999)
  {
    fprintf(stderr,"Bad NMec (1 <= NMex (%d) <= 999999)\n",NMec);
    exit(1);
  }
  if(T < 1 || T > MAX_T)
  {
    fprintf(stderr,"Bad T (1 <= T (%d) <= %d)\n",T,MAX_T);
    exit(1);
  }
  if(P < 1 || P > MAX_P)
  {
    fprintf(stderr,"Bad P (1 <= P (%d) <= %d)\n",P,MAX_P);
    exit(1);
  }
  //
  // the starting and ending dates of each task satisfy 0 <=
starting_date <= ending_date <= total_span
  //
  total_span = (10 * T + P - 1) / P;
  if(total_span < 30)
    total_span = 30;
  //
  // probability of each possible task duration
  //
  // task span relative probabilities
  //
  // |  0  0  4  6  8 10 12 14 16 18 | 20 | 19 18 17 16 15 14 13 12 11
10  9  8  7  6  5  4  3  2  1 | smaller than 1
  // |  0  0  2  3  4  5  6  7  8  9 | 10 | 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 | 30 31 ... span
  //
  weight = (int *)alloca((size_t)(total_span + 1) * sizeof(int)); //
allocate memory (freed automatically)
  if(weight == NULL)
  {
    fprintf(stderr,"Strange! Unable to allocate memory\n");
    exit(1);
  }
```

```
#define sum1  (298.0)                       // sum of weight[i] for
i=2,...,29 using the data given in the comment above
#define sum2  ((double)(total_span - 29))  // sum of weight[i] for
i=30,...,data_span using a weight of 1
#define tail  100
  scale = (int)ceil((double)tail * 10.0 * sum2 / sum1); // we want that
scale*sum1 >= 10*tail*sum2, so that large task
  if(scale < tail)                                    // durations
occur 10% of the time
    scale = tail;
  weight[0] = 0;
  weight[1] = 0;
  for(i = 2;i <= 10;i++)
    weight[i] = scale * (2 * i);
  for(i = 11;i <= 29;i++)
    weight[i] = scale * (30 - i);
  for(i = 30;i <= total_span;i++)
    weight[i] = tail;
#undef sum1
#undef sum2
#undef tail
  //
  // accumulate the weigths (cummulative distribution)
  //
  for(i = 1;i <= total_span;i++)
    weight[i] += weight[i - 1];
  //
  // generate the random tasks
  //
  srandom(NMec + 314161 * T + 271829 * P);
  problem->NMec = NMec;
  problem->T = T;
  problem->P = P;
  problem->I = (ignore_profit == 0) ? 0 : 1;
  for(i = 0;i < T;i++)
  {
    //
    // task starting an ending dates
    //
    r = 1 + (int)random() % weight[total_span]; // 1 ..
weight[total_span]
    for(span = 0;span < total_span;span++)
      if(r <= weight[span])
        break;
```

```
    problem->task[i].starting_date = (int)random() % (total_span - span
+ 1);
    problem->task[i].ending_date = problem->task[i].starting_date + span
- 1;
    //
    // task profit
    //
    // the task profit is given by r*task_span, where r is a random
variable in the range 50..300 with a probability
    //   density function with shape (two triangles, the area of the
second is 4 times the area of the first)
    //
    //      *
    //     /|   *
    //    / |       *
    //   /  |           *
    //  *---*---------------*
    // 50 100 150 200 250 300
    //
    scale = (int)random() % 12501; // almost uniformly distributed in
0..12500
    if(scale <= 2500)
      problem->task[i].profit = 1 + round((double)span * (50.0 +
sqrt((double)scale)));
    else
      problem->task[i].profit = 1 + round((double)span * (300.0 - 2.0 *
sqrt((double)(12500 - scale))));
  }
  //
  // sort the tasks by the starting date
  //
  qsort((void
*)&problem->task[0],(size_t)problem->T,sizeof(problem->task[0]),compare_
tasks);
  //
  // finish
  //
  if(problem->I != 0)
    for(i = 0;i < problem->T;i++)
      problem->task[i].profit = 1;
#define DIR_NAME  problem->dir_name
  if(snprintf(DIR_NAME,sizeof(DIR_NAME),"%06d",NMec) >=
sizeof(DIR_NAME))
  {
```

```
    fprintf(stderr,"Directory name too large!\n");
    exit(1);
  }
#undef DIR_NAME
#define FILE_NAME  problem->file_name

if(snprintf(FILE_NAME,sizeof(FILE_NAME),"%06d/%02d_%02d_%d.txt",NMec,T,P
,problem->I) >= sizeof(FILE_NAME))
  {
    fprintf(stderr,"File name too large!\n");
    exit(1);
  }
#undef FILE_NAME
}

#endif


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//
//TODO problem solution (place your solution here)
void awesome_implementation(problem_t *problem) {
  int k, combination_profit,max_profit,tasks_done,max_n_of_tasks,
task_idx, retval;
  long long i,j,viable_job_selections;

  max_profit = 0;
  viable_job_selections = pow(2, problem->T);
  max_n_of_tasks = 0;
  FILE *out=fopen("viable_job_selections_profits.txt","w");

  for (i = 0; i < 1LL << (long long) problem->T; ++i)//ESTE FOR É
CHAMADO 2^LEN -1 VEZES
  {
    memset(problem->busy, -1, problem->P * sizeof(int)); //limpar o
array busy para cada combinação
    combination_profit = 0;
    tasks_done = 0;
    task_idx = 0;
    retval = 0;
    for (j = problem->T - 1; j >= 0; --j) //este for gera cada bit da
combinação
    {
```

```
      if (((1LL << j) & i) > 0LL) { //((1<<j)&i) > 0 evaluates to 1 if
the jth bit is set, 0 otherwise
        problem->task[task_idx].assigned_to = -1;
        for (k = 0; k < problem->P; k++)
        {
          if (problem->task[task_idx].starting_date > problem->busy[k])
          { //verificar se o programador está disponivel para trabalhar
nessa data
            problem->task[task_idx].assigned_to = k;   //atribuir a
tarefa ao P
            problem->busy[k] = problem->task[task_idx].ending_date;
//atualizar a disponibilidade do P
            combination_profit = combination_profit +
problem->task[task_idx].profit;
            tasks_done++;
            break; // terminar o ciclo for
          }
        }
        if (problem->task[task_idx].assigned_to == -1) {
          retval = 1;
        }
      }
      task_idx++;
    }
    if (combination_profit > max_profit) //se a combinação for viavel,
comparar o profit
    {
      max_profit = combination_profit;
    }
    if (tasks_done > max_n_of_tasks) {
      max_n_of_tasks = tasks_done;
    }
    if (retval) { //Se a tarefa é para ser feita e não for atribuida
      viable_job_selections--;
    } else {
      fprintf(out, "%d\n", combination_profit);
    }
  }
  problem->total_profit = max_profit;
  printf("VIABLE JOB SELECTIONS: %lld\nPROFIT: %d\nLARGEST NUMBER OF
PROGRAMMING TASKS: %d\n", viable_job_selections, problem->total_profit,
max_n_of_tasks);
  fclose(out);
}
```

```
#if 1

static void solve(problem_t *problem)
{
  FILE *fp;
  int i;

  //
  // open log file
  //
  (void)mkdir(problem->dir_name,S_IRUSR | S_IWUSR | S_IXUSR);
  fp = fopen(problem->file_name,"w");
  if(fp == NULL)
  {
    fprintf(stderr,"Unable to create file %s (maybe it already exists?
If so, delete it!)\n",problem->file_name);
    exit(1);
  }
  //
  // solve
  //
  problem->cpu_time = cpu_time();
  awesome_implementation(problem);//É CHAMADA A FUNCAO AQUI
  problem->cpu_time = cpu_time() - problem->cpu_time;
  //
  // save solution data
  //
  fprintf(fp,"NMec = %d\n",problem->NMec);
  fprintf(fp,"T = %d\n",problem->T);
  fprintf(fp,"P = %d\n",problem->P);
  fprintf(fp,"Profits%s ignored\n",(problem->I == 0) ? " not" : "");
  fprintf(fp,"Solution time = %.3e\n",problem->cpu_time);
  fprintf(fp,"Task data\n");
#define TASK  problem->task[i]
  for(i = 0;i < problem->T;i++)
    fprintf(fp,"  %3d %3d
%5d\n",TASK.starting_date,TASK.ending_date,TASK.profit);
#undef TASK
  fprintf(fp,"End\n");
  //
  // terminate
  //
  if(fflush(fp) != 0 || ferror(fp) != 0 || fclose(fp) != 0)
  {
```

```
    fprintf(stderr,"Error while writing data to file
%s\n",problem->file_name);
    exit(1);
  }
}

#endif



////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//
// main program
//

int main(int argc,char **argv)
{
  problem_t problem;
  int NMec,T,P,I;

  NMec = (argc < 2) ? 2020 : atoi(argv[1]);
  T = (argc < 3) ? 5 : atoi(argv[2]);
  P = (argc < 4) ? 2 : atoi(argv[3]);
  I = (argc < 5) ? 0 : atoi(argv[4]);
  init_problem(NMec,T,P,I,&problem);
  solve(&problem);

  return 0;
}
```