universidade de aveiro

Algoritmos e Estruturas de Dados

Trabalho Prático 1

2020/2021

Diana Elisabete Siso Oliveira, nº 98607, P2 (33.3%)
Miguel Rocha Ferreira, nº 98599, P2 (33.3%)
Paulo Guilherme Soares Pereira, nº 98430, P2 (33.3%)

Índice

1. Introdução

Com o desenvolvimento do presente projeto, visamos aprimorar as nossas capacidades de programação em C e de desenvolvimento e entendimento do funcionamento de algoritmos de ordenação e filtragem de conteúdo.

Para tal, através de informações fornecidas previamente no enunciado e dos conhecimentos adquiridos ao longo do semestre na respetiva unidade curricular, é esperado que o resultado final do *script* possua algoritmos eficientes, de forma a atender às expectativas do projeto.

O conteúdo do enunciado centra-se, portanto, na atribuição de tarefas a programadores, tendo em conta que nem sempre todas as tarefas poderão ser designadas ou nem sempre todos os programadores ficarão com uma tarefa designada, sendo necessário encontrar a melhor forma dessa atribuição ser realizada escolhendo um dos seguintes dois parâmetros: pretendemos fazer o máximo número de tarefas ou obter o máximo lucro possível nas tarefas que escolhemos?

Assim, contemplando as ideias base do enunciado, foram desenvolvidas várias implementações, cada uma correspondente a uma diferente abordagem ao problema, sendo possível observar a correção das nossas falhas ao longo do desenvolvimento do código do *script*, de forma a imprimir num ficheiro final de texto o resultado da melhor combinação para o parâmetro desejado num menor intervalo de tempo.

2. Explicação do código fornecido e adaptações

Começaremos por uma breve explicação do código fornecido no enunciado do presente projeto, mais especificamente no ficheiro *job_selection*. Para um melhor entendimento, a explicação encontra-se dividida em secções, a cada secção corresponde um sub-tópico.

2.1 Estruturas

Ao analisar o código presente no ficheiro anteriormente mencionado, é possível observar a existência de duas estruturas fundamentais na resolução do projeto: *task_t* e *problem_t*.

Na estrutura *task_t* somo capazes de observar diversos campos que correspondem a atributos que uma determinada tarefa pode possuir, nomeadamente a data inicial, a data final, o *profit* (ou o lucro da tarefa) e ainda um inteiro, *assigned_to*, que define a qual programador a tarefa foi designada num dado instante.

Já a estrutura *problem_t* contém os atributos que definem o problema, nomeadamente os *inputs* Nmec (número mecanográfico), T (número de tarefas), P (número de programadores), I (decisão de ignorar o lucro ou não). Esses *inputs* são, portanto, responsáveis por gerar um problema com P programadores e T tarefas, ignorando ou não os lucros (valor de I igual a 1 ou 0, respectivamente). Atente-se que o problema é gerado de forma pseudo aleatória e não totalmente aleatória, e assim é possível a comparação de resultados quando os mesmos *inputs* são usados.

Nesta estrutura existe ainda um inteiro capaz de guardar o lucro total num determinado instante (*total_profit*); um *double cpu_time* que representará o tempo que foi usado para encontrar a solução do problema atual; e dois *arrays*: um *array* de tarefas, onde são armazenadas as informações das tarefas geradas e um *array busy*, que armazena até quando cada programador se encontra ocupado (tendo o valor -1 quando ele se encontra livre). Por fim, existem ainda dois atributos que definem o nome do diretório e o nome do ficheiro onde serão guardadas as informações da solução do problema.

2.2 Funções

Ao longo do ficheiro é possível encontrar várias funções que irão ser úteis na realização das próximas implementações: *compare_tasks*, *init_problem*, *solve* e *main*. Da função *compare_tasks* surgiram mais duas funções: *compare_tasks_ending* e *compare_tasks_ending_2*.

A função *compare_tasks* tem como objetivo comparar duas tarefas que foram dadas como argumentos, de modo a, no final, obter um *array* de tarefas ordenado por ordem crescente da data inicial.

Para comparar as tarefas, a função começa por verificar a sua data de início; caso a primeira tarefa a ser introduzida (t1) tenha uma data inicial inferior à segunda (t2) devolve 1; caso a primeira tarefa a ser introduzida (t1) tenha uma data inicial superior à segunda (t2) devolve -1; caso se verifique que ambas têm a mesma data de início é então verificada a data de conclusão. Se a tarefa t1 acabar antes da tarefa t2 é devolvido 1; se a tarefa t1 acabar depois da tarefa t2 é devolvido o valor -1; se ambas as tarefas tiverem a mesma data de início e de conclusão é devolvido o valor 0.

```
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;
}
```

A função *compare_tasks_ending* funcionará seguindo uma metodologia semelhante, porém o seu objetivo é organizar as tarefas por ordem crescente da data de conclusão. Já a função *compare_tasks_ending_2* irá ordenar as tarefas por ordem decrescente da data final de conclusão.

```
int compare_tasks_ending_2(const void *t1,const void *t2){
   int d1,d2;
   d1 = ((task_t *)t1)->ending_date;
   d2 = ((task_t *)t2)->ending_date;
   if(d1 != d2)
         return (d1 < d2) ? -1 : +1;
   d1 = ((task_t *)t1)->starting_date;
   d2 = ((task_t *)t2)->starting_date;
   if(d1 != d2)
         return (d1 < d2) ? -1 : +1;
   return 0;
}
```

```
int compare_tasks_ending(const void *t1,const void *t2){
   int d1,d2;
   d1 = ((task_t *)t1)->ending_date;
   d2 = ((task_t *)t2)->ending_date;
   if(d1 != d2)
         return (d1 > d2) ? -1 : +1;
   d1 = ((task_t *)t1)->starting_date;
   d2 = ((task_t *)t2)->starting_date;
   if(d1 != d2)
         return (d1 > d2) ? -1 : +1;
   return 0;
}
```

A função *init_problem* tem a funcionalidade de verificar se os *inputs* são válidos bem como o período que as tarefas duram (total span). Além disso, a função é responsável também por reservar a memória necessária para o problema.

A função *solve*, por sua vez, contém as resoluções do problema proposto para este projeto e armazena as informações relativas às soluções num ficheiro - verificando, previamente, se é possível ou não criar esse ficheiro.

Por último, a função *main* do ficheiro guarda nas respetivas variáveis os valores introduzidos no terminal quando executamos o *script*, tendo como valores *default* 2020 para o número mecanográfico (NMec), 5 para o número de tarefas (T), 2 para o número de programadores (P) e 0 para a decisão de ignorar ou não ignorar o lucros (I), para além de chamar as funções de inicialização do problema e da sua resolução.

Adicionamos ainda à função *main* um conjunto de condições *if* como forma de validação do *input* introduzido no terminal referente à decisão de ignorar ou não os lucro. Se o *input* não corresponder nem ao valor 0 nem ao valor 1, então impressa uma mensagem de erro e o programa tem a sua execução interrompida. Caso o valor de I seja igual a 0, o utilizador poderá escolher entre 2 implementações realizadas - se escolher um número diferente de 1 ou 2 então será impressa uma mensagem de erro e o programa tem a sua execução interrompida. Caso o valor de I seja igual a 1, o utilizador poderá escolher entre 5 implementações realizadas - se escolher um número diferente de 1, 2, 3, 4 ou 5 então será impressa uma mensagem de erro e o programa tem a sua execução interrompida.

```
 if (I == 1) {
    int option;
     printf("Você escolheu ignorar os lucros! Temos 5 implementações que você poderá escolher!\n(1) SEGUNDA ABORDAGEM\n(2) TERCEIRA ABORDAGEM\n(3) QUARTA ABORDAGEM\n(4) QUINTA ABORDAGEM\n(5) SEXTA ABORDAGEM\nInsira um dos 5 números: \n");
    scanf("%d", &option);
```

```
          if ((option == 1) ||  (option == 2) || (option == 3) || (option == 4) || (option
== 5)){
                init_problem(NMec,T,P,I,&problem, option);
                solve(&problem, option);
          }
          else {
                printf("Opção inválida!");
                return EXIT_FAILURE;
          }
       }
       else if (I == 0) {
          int option;
           printf("Você escolheu não ignorar os lucros! Temos 3 implementações que você
poderá escolher!\n(1) PRIMEIRA ABORDAGEM\n(2) SEGUNDA ABORDAGEM\nInsira um dos 3 números:
\n");
          scanf("%d", &option);
          if ((option == 1) || (option == 2)){
                init_problem(NMec,T,P,I,&problem, option);
                solve(&problem, option);
          }
          else {
                printf("Opção inválida!");
                return EXIT_FAILURE;
          }
       }
       else {
          printf("O valor do I é inválido! Escolha 1 para ignorar os profits ou 0 para não
ignorar os profits!\n");
       }
```

3. Exposição das abordagens (ignorando o lucro)

Neste tópico iremos apenas focar nas abordagens utilizadas para resolver o enunciado referente à procura da melhor combinação de forma a que os programados realizem o máximo número de tarefas. Nem todas as abordagens que serão explicadas apresentam uma implementação correta - apenas as últimas duas abordam o problema de forma correta -, no entanto achamos importante referi-las, pois elas são um medidor da nossa evolução ao longo do presente projeto e, algumas delas, revelaram-se abordagens eficientes seguindo determinadas limitações.

Cada abordagem, com exceção da primeira mencionada, será acompanhada de dois gráficos que traduzem o tempo de execução do *script* em função do número de tarefas para um número de programadores fixo (2 programadores) e o tempo de execução do *script* em função do número de programadores para um número de tarefas fixo (20 tarefas). Esses gráficos serão analisados num só no último tópico desta secção.

3.1 Primeira Abordagem

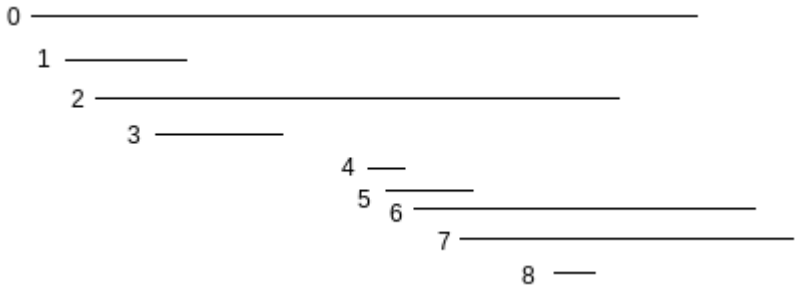
Inicialmente abordamos o problema seguindo um raciocínio que se mostrou funcional apenas para um número máximo de 1 programador. Isto porque o raciocínio consistia em atribuir ao primeiro programador a combinação com o maior número de tarefas que ele podia fazer, porém sem verificar se existe outra combinação mais benéfica tendo em conta que existem mais programadores ainda sem tarefa atribuídas, logo essa lógica não é correta. Qual das seguintes combinações é mais benéfica para o problema apresentado?

a) Programador 1 - 4 tarefas; Programador 2 - 2 tarefas; Programador 3 - 2 tarefas
b) Programador 1 - 4 tarefas; Programador 2 - 3 tarefas; Programador 2 - 2 tarefas

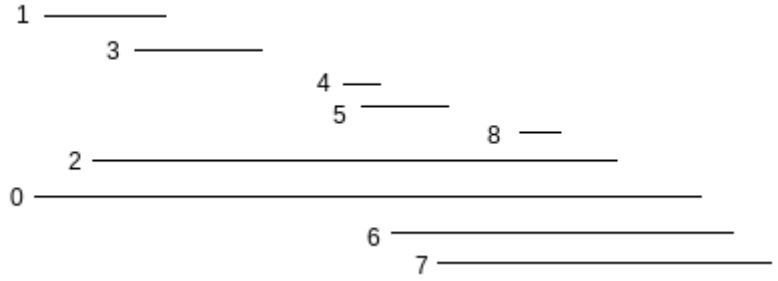
Obviamente, a melhor combinação é a b). Por vezes é melhor que o nosso primeiro programador não faça uma determinada combinação, mas sim outra com mesmo número de tarefas, pois isso pode implicar que ele faça uma determinada tarefa que, tendo em conta o intervalo de tempo das *tasks* que sobram, era melhor se essa tarefa fosse feita por um programador seguinte.

No entanto, essa consequência negativa apenas é visível a partir de um determinado número de programadores; para 1 programador a consequência nem será aplicada e, portanto, o código mostra-se eficiente.

Para além disso, o raciocínio que aplicamos começava por ordenar as tarefas por ordem crescente de data final, como explicado no gráfico abaixo:



Ordem das tarefas seguindo a ordenação mencionada: 1 3 4 5 8 2 0 6 7



Desta forma, as tarefas que terminam primeiro ficam numa posição mais perto da inicial e tarefas que possuem uma data inicial baixa mas uma data de término elevada dificilmente serão escolhidas, pois apenas uma passagem pelo *array* já nos daria o máximo número de tarefas que **1** programador é capaz de realizar. Por exemplo, na imagem mostrada, a tarefa 1 é a que possui menor data de término, pelo que será a primeira que o programador faria; de seguida ele faria a tarefa 3, a tarefa 4 e a tarefa 5. A tarefa 2 e a tarefa 0, apesar de começarem cedo, possuem uma duração muito longa, pelo que não é viável escolhê-las, então não será necessário realizar um *loop for* para escolher a melhor combinação, a primeira combinação será a melhor combinação para 1 programador.

Apesar de, numa etapa inicial, o código não se mostrar útil para a resolução do problema quando o número de programadores é superior a 1, posteriormente tentamos melhorá-lo adicionando um condição de verificação de melhor combinação, pelo que iremos explicar o código final desta abordagem no tópico 3.4 Quarta Abordagem.

3.2 Segunda Abordagem

Nesta implementação pretendemos saber qual o máximo de tarefas que é possível fazer tendo em conta o número de programadores e as datas de realização das tarefas.

É de notar que cada programador pode realizar apenas uma tarefa de cada vez e que tem que levar a mesma até ao fim.

Para a resolução deste *assignment problem* vamos testar várias combinações de realização de tarefas e ver qual resulta no maior número de tarefas realizadas.

Começamos por inicializar o vetor *problem->busy*, que tem a dimensão do número de programadores, com todas as posições a -1 e também o vetor *assigned_to*, que contém o programador ao qual uma certa tarefa está atribuída, a -1.

```
for (int i=0;i<problem->P;i++)
 {
   problem->busy[i]=-1;
 }


  for(int i=0;i<problem->T;i++)
 {
   problem->task[i].assigned_to=-1;
 }
```

Na função solve, começamos por iterar pelo número de programadores (problem->P) .

Para cada programador, as variáveis *numeroTasksProgramador* (número de tarefas que o programador vai fazer em cada iteração) e *numeroTasksTotal* (melhor número de tarefas que cada programador vai fazer ao longo de todas as iterações) são igualadas a zero.

Para cada programador é também impressa uma linha no ficheiro de saída com o conteúdo "PROGRAMADOR X" em que 'X' é o índice do programador em questão.

```
for(int p=0; p<problem->P; p++)
{
     fprintf(fp,"\nPROGRAMADOR %d\n",(p+1));
     numeroTasksProgramador=0;
     numeroTasksTotal=0;
```

Dentro de cada iteração pelos programadores, iteramos também pelo número de tarefas (*problem->T*). Cada valor de t ditará a tarefa de início para cada programador, ou seja, quantas tarefas pode o programador fazer se começar pela tarefa 0, quantas pode fazer se começar pela tarefa 1, se começar pela tarefa 2 ... até à última tarefa.

No início de cada iteração pelas tarefas, chamamos a função *funcaoTask* e atribuímos o seu valor de retorno à variável *numeroTasksProgramador*.

```
numeroTasksProgramador=funcaoTask(problem,t,p,numeroTasksTotal);
```

Passamos agora ao funcionamento da função funcaoTask.

Esta função tem como argumentos de entrada um ponteiro para a variável *problem* do tipo *problem_t*, um *int tarefa* (que é o índice da iteração pelas tarefas descrita anteriormente), um *int programador* (que é o índice da iteração pelos programadores descrita anteriormente) e um *int nrTasksTotal* (que é melhor número de tarefas que cada programador vai fazer ao longo de todas as iterações, como já foi referido anteriormente).

Por cada chamada à função *funcaoTask*, inicializamos o vetor *busy* todo a zeros e alocamos espaço para o array *tarefasProgramador* que terá o tamanho *problem->T*. Após a alocação de espaço, preenchemos o vetor todo com o valor -1.

```
for(int n=0; n<problem->P; n++)
 {
   problem->busy[n]=-1;
 }

 int *tarefasProgramador= (int *) malloc(sizeof(int)*problem->T);

 for(int m=0; m<problem->T;m++)
 {
   tarefasProgramador[m]=-1;
 }
```

Começamos por inicializar a variável nrTasks a 0. Esta variável é inicializada a zero sempre que a função é chamada e vai guardar o número de tarefas que o programador em questão faz em cada chamada à função.

Agora iteramos pelas tarefas, sendo o início da iteração marcada pelo argumento de entrada *tarefa*, que é dado por cada iteração do ciclo *for* presente na função *solve*.

Caso o programador esteja disponível (o valor de *busy* com o seu índice tenha o valor -1) e caso a tarefa em questão não esteja atribuída (o seu valor *assigned_to* seja igual a -1), a tarefa vai ser atribuída ao programador e então guardamos essa tarefa no vetor *tarefasProgramador*. Incrementamos também o valor de *nrTasks* e colocamos o valor da *ending_date* da tarefa no vetor *busy*, na posição referente ao mesmo, deixando assim o programador ocupado até que a tarefa acabe.

```
int nrTasks=0;
 for(int k=tarefa; k<problem->T;k++)
 {
   if(problem->busy[programador]==-1)
```

```
    {
      if(problem->task[k].assigned_to==-1)
      {
        tarefasProgramador[nrTasks]=k;
        nrTasks++;
problem->busy[programador]=problem->task[k].ending_date;
      }
    }
```

Caso o programador não esteja imediatamente disponível, ou seja, se já tiver uma tarefa atribuída, verificamos também se a tarefa já foi atribuída a outro programador. Caso não esteja, vemos se o programador ainda está ocupado. Para isso, verificamos o valor de *busy* para este programador específico. Caso o valor de *busy* seja menor que o valor da *starting_date* da tarefa em questão, quer dizer que pode ser realizada, visto que a tarefa que o programador está a fazer vai acabar antes do início da atual.

Sendo assim, realiza-se o mesmo que na anterior, a tarefa vai ser atribuída ao programador e então guardamos esta tarefa no vetor *tarefasProgramador*. Incrementamos também o valor de *nrTasks* e colocamos o valor da *ending_date* da tarefa no vetor *busy*, na posição referente ao mesmo, deixando assim o programador ocupado até que a tarefa acabe.

```
else
    {
      if(problem->task[k].assigned_to==-1)
      {
         if(problem->task[k].starting_date>problem->busy[programador])
        {
         problem->busy[programador]=problem->task[k].ending_date;
          tarefasProgramador[nrTasks]=k;
          nrTasks++;
        }
      }
  }
```

De seguida vamos verificar se a tarefa feita nesta iteração é compatível com alguma tarefa anterior que não tenha sido realizada. Para isso vamos iterar pelas tarefas anteriores à tarefa em questão, de forma decrescente, até chegarmos à tarefa 0.

Começamos por definir o valor de *busy* como o valor da *starting_date* da tarefa em questão, visto que queremos apenas encontrar uma tarefa cuja *ending_date* seja menor que a *starting_date* da atual. A seguir iteramos pelas tarefas, sendo a tarefa de cada iteração definida pela variável *element*. A lógica é a mesma da anterior, caso a tarefa não esteja atribuída e as suas *starting_date*'s e *ending_date*'s não se sobreponham, a tarefa pode ser realizada e vai ser guardada no vetor *tarefasProgramador*. O valor de *busy* será agora definido para o valor da *starting_date* desta tarefa e o *nrTasks* é incrementado.

```
problem->busy[programador] = problem->task[tarefa].starting_date;
 for (int element = tarefa; element>0; element--) {
     if(problem->task[element].assigned_to==-1) {
         if (problem->task[element].ending_date<problem->busy[programador]) {
             tarefasProgramador[nrTasks]=element;
             problem->busy[programador]=problem->task[element].starting_date;
             nrTasks++;
         }
     }
 }
```

Caso o número de tarefas realizadas nesta iteração seja maior que o máximo número de tarefas realizadas até ao momento (argumento de entrada), iteramos por todas as tarefas e caso o valor de *assigned_to* tenha o valor do programador em questão vamos eliminar este registo, visto que já não é a solução ideal. Para isso passamos estes valores de novo a -1.

O número máximo de tarefas passa agora a ser o número de tarefas desta iteração.

Agora para cada tarefa, enquanto o vetor *tarefasProgramador* tiver valores diferentes de -1 (tiver tarefas) vamos atribuir essas tarefas ao programador em questão, alterando o valor do *assigned_to*.

```
if(nrTasks>nrTasksTotal)
   {
   for(int i=0;i<problem->T;i++)
   {
     if(problem->task[i].assigned_to==programador)
     {
       problem->task[i].assigned_to=-1;
     }
   }
   nrTasksTotal=nrTasks;
   for(int i=0; i<problem->T;i++)
   {
     if(tarefasProgramador[i]!=-1)
     {
       problem->task[tarefasProgramador[i]].assigned_to=programador;
     }
   }
 }
```

No final, a função retorna o número de tarefas realizadas nesta iteração.

Fora da função voltamos a verificar se o número de tarefas devolvido pela função é superior ou não ao número máximo de tarefas. Caso o valor seja maior, o valor máximo vai ser atualizado.

```
if(numeroTasksProgramador>numeroTasksTotal)
       {
         numeroTasksTotal=numeroTasksProgramador;
```

```
        }
```

Agora, para imprimirmos os resultados no ficheiro, iteramos pelas tarefas e caso a tarefa da iteração tenha como valor *assigned_to* o número do programador, adicionamos uma linha de texto ao ficheiro a dizer que a tarefa da iteração foi atribuída.

```
    for(int i=0;i<problem->T;i++)
        {
          if(problem->task[i].assigned_to==p)
          {
              fprintf(fp, "Tarefa que começa em %d e acaba em %d\n",
problem->task[i].starting_date, problem->task[i].ending_date);
          }
        }
```

A seguir contabilizamos quantas tarefas foram feitas no total e imprimimos também no ficheiro este valor.

```
    for(int g=0; g<problem->T; g++)
       {
         if(problem->task[g].assigned_to!=-1)
         {
           numeroTasksTodas++;
         }
       }
       fprintf(fp,"\nNúmero de Tasks Realizadas: %d\n", numeroTasksTodas);
```

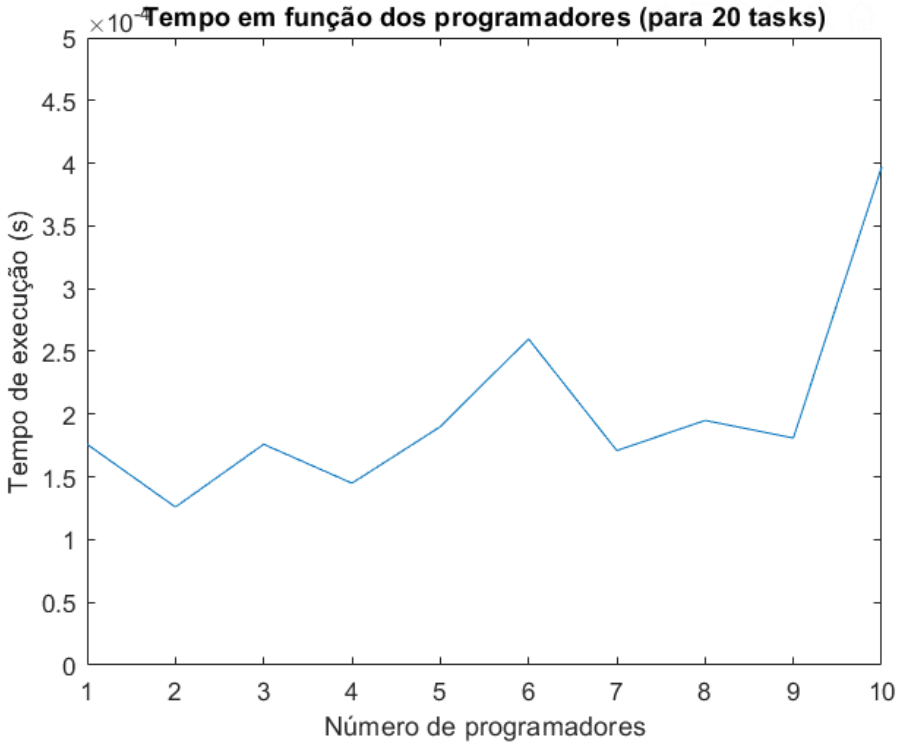


Fig.01 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

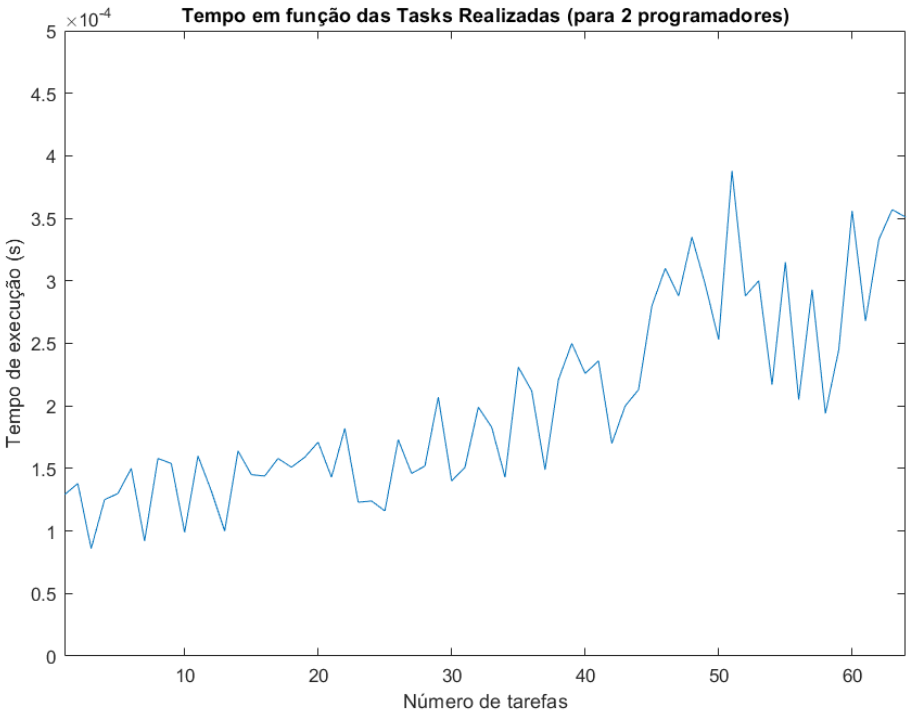


Fig.02 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 64 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

3.3 Terceira Abordagem

Temos, também, uma terceira abordagem funcional que interpreta o problema apresentado de maneira diferente: utilizando uma matriz de compatibilidade.

O código começa com a inicialização da matriz, *MatrizCompativeis*, em que tanto as linhas como as colunas representam as tarefas existentes ordenadas por ordem crescente de data inicial - foi previamente configurado no programa que, se o *input* do utilizador no terminal for 2, a ordenação das tarefas vai se dar seguindo esse critério.

```
int **MatrizCompativeis;
MatrizCompativeis = malloc(problem->T * sizeof(int*));
for (int i=0; i<problem->T; i++) {
MatrizCompativeis[i] = malloc(problem->T * sizeof(int));
}
```

Essa matriz, como o próprio nome indica, servirá para guardar a informação de quais tarefas são compatíveis, sendo, portanto, uma *matriz simétrica*. Se a tarefa x é compatível com a tarefa y, então na linha x e na coluna y será colocado o número 1, assim como na linha y e na coluna x será colocado o mesmo número; caso não se verifique compatibilidade entre a tarefa x

e a tarefa y, na linha x e na coluna y será colocado o número 0, assim como na linha y e na coluna x será colocado o número 0. Para haver compatibilidade, tendo em conta que as tarefas estão ordenadas por ordem crescente de data inicial, é necessário que a data inicial da tarefa y seja *maior* que a data final da tarefa x, ou, sinonimamente, que a data final da tarefa x seja *menor* que a data inicial da tarefa y. Para efeitos próximos, optamos por manter a diagonal da matriz com valores 1.

```
for (int x=0; x<problem->T; x++) {
   for (int y=x; y < problem->T; y++) {
         if (x==y) {
               MatrizCompativeis[x][y] = 1; MatrizCompativeis[y][x] = 1;
         }
         else{
               if (problem->task[x].ending_date < problem->task[y].starting_date) {
                     MatrizCompativeis[x][y] = 1; MatrizCompativeis[y][x] = 1;
               }
               else {
                     MatrizCompativeis[x][y] = 0; MatrizCompativeis[y][x] = 0;
               }
         }
    }
}
```

A matriz apresentada será o principal motor para o código correspondente a esta abordagem e irá ser modificada ao longo do programa à medida que as tarefas vão sendo atribuídas aos programadores. Feita a inicialização da matriz, entramos num *loop for* que percorre todos os programas designados para o presente problema.

O primeiro passo é atribuir o valor 0 às variáveis *totalTasks* e *tasksValidas*, cuja funcionalidade posteriormente será ditar qual é o número máximo de tarefas que o programador pode realizar *até ao momento* (ou seja, os seus valores vão sendo alterados ao longo do código) e inicializamos um vetor denominado *tarefasProgramador*, onde serão guardadas essas tarefas, e outro vetor denominado *jaEscolhida*, cuja funcionalidade será explicada em breve.

```
for(int p=0; p<problem->P; p++) {
int totalTasks = 1;
int *tarefasProgramador=(int *) malloc(sizeof(int)*(problem->T));
int *jaEscolhida=(int *) malloc(sizeof(int)*(problem->T));
for(int m=0; m<problem->T;m++) {
tarefasProgramador[m]=-1;
}
. . .
}
```

Dado este ponto, precisamos então de saber quais são as tarefas que irão ser atribuídas ao programador, o que nos leva a entrar num *loop for* para percorrê-las. Dentro desse ciclo,

atribuímos a todos os espaços do vetor *jaEscolhida* o valor -1, vetor esse que servirá para armazenar as tarefas que forem escolhidas para uma combinação possível ao longo do ciclo, de forma a, sempre que introduzirmos um nova tarefa, validarmos primeiro se a mesma é compatível com todas as que foram escolhidas previamente. Seguidamente, invocamos a função *MelhorComb* guardando o seu valor de retorno na variável *tasksValidas*.

```
    for (int tarefa=0; tarefa<problem->T; tarefa++) {
    for (int i=0; i<problem->T; i++) { jaEscolhida[i]=-1;}
    tasksValidas = MelhorComb(tarefa, MatrizCompativeis, problem, jaEscolhida,
tarefasProgramador, totalTasks);
       . . .
    }
```

A função *MelhorComb* tem como objetivo encontrar a melhor combinação de tarefas para o programador realizar. Como tal, já tendo a tarefa inicial da combinação correspondente à iteração definida previamente, iremos entrar num *loop for* que percorrerá todas as tarefas; ao encontrar uma que seja compatível, isto é, que na matriz *MatrizCompativeis* tenha associado o valor 1 na linha correspondente à tarefa que passamos como argumento na chamada da função e na coluna correspondente à tarefa analisada no ciclo *for*, a variável *flag* assume o valor 0. (1)

Encontramos uma tarefa que seja compatível com a tarefa que invocou a função, mas será que ela é compatível com as tarefas que foram encontradas anteriormente, se já tiverem sido encontradas mais tarefas, e que atualmente já constam no vetor *jaEscolhida*? Analisamos essa condição através de um outro ciclo *for*. Nesse ciclo, interessa-nos apenas ir até o valor atual do *counter*, pois tudo o que for maior que esse valor, no vetor *jaEscolhida* tem o valor -1 associado.

Procedemos então a uma filtragem de forma a validar se a tarefa em análise não se sobrepõe com nenhuma tarefa já capturada previamente. Caso haja sobreposição, o valor associado à *flag* mudará para 1 e sairemos do *loop for*, pois já não nos interessa continuar a comparar.

```
    static int MelhorComb(int tarefa, int **MatrizCompativeis, problem_t *problem, int
*jaEscolhida, int *tarefasProgramador, int totalTasks) {
        int counter = 1;
        jaEscolhida[0] = tarefa;
        for (int element=0; element<problem->T; element++) {
            if(MatrizCompativeis[tarefa][element] == 1) {
                int flag=0;
                for (int e2=0; e2 <(counter); e2++) {
                    if ((MatrizCompativeis[jaEscolhida[e2]][element] != 1) ||
(jaEscolhida[e2] == element)) {
                        flag = 1; break;
                    }
                }
            }
            . . .
```

```
}
```

Realizada a filtragem, se o valor da *flag* tiver sido alterado para 1 em qualquer momento, significa que a tarefa analisada não é uma opção viável, pelo que passamos para a próxima tarefa do ciclo *for*. Caso o valor da *flag* tenha permanecido 0, a tarefa analisada é compatível com todas as outras previamente escolhidas e, por isso, adicionamo-la ao vetor *jaEscolhida* e incrementamos a variável *counter*, passando para a próxima tarefa do ciclo.

```
if (flag == 1) { continue; }
if (flag == 0){ //printf("Sou compatível com a %d!\n", element);
   jaEscolhida[counter] = element;
   counter++;
}
```

(1) Caso a tarefa analisada não seja compatível, então irá avançar para a próxima tarefa do ciclo através da instrução *continue*.

No final do *loop for*, tendo em conta que a função *MelhorComb* será chamada para cada tarefa existente de modo a ter combinações que comecem com 0, com 1, com 2, até à última tarefa existente, e que o *counter* vai sendo incrementado cada vez que uma nova *task* é adicionada ao *array jaEscolhida*, se o valor do *counter* for maior que o valor da variável *totalTasks* - lembre-se que essa variável traduz o número máximo de tarefas que é possível fazer *até ao momento* -, significa que existe uma combinação em que é possível fazer mais tarefas do que era possível anteriormente, logo, o valor associado à variável *totalTasks* será mudado para o valor do *counter* e copiaremos os elementos do vetor *jaEscolhida* que sejam diferentes de -1 (ou seja, limitamo o ciclo *for* até *counter*) para o *array tarefasProgramador* depois de nos assegurarmos que a informação previamente contida neste vetor era apenas valores -1. Por último, retornamos o valor da *counter*.

```
   if (totalTasks < counter) {
         for (int x=0; x<problem->T; x++) {
               tarefasProgramador[x] = -1;
         }
         totalTasks = counter;
         for (int x=0; x<problem->T; x++) {
               tarefasProgramador[x] = jaEscolhida[x];
         }
    }
    return counter;
}
```

Após a invocação da função *MelhorComb*, teremos então a melhor combinação de tarefas para um *array* que comece na tarefa correspondente à iteração do ciclo *for*, porém essa combinação pode ser pior comparada com uma que comece na tarefa da iteração seguinte. Para

capturarmos a melhor combinação entre todas as combinações boas geradas no ciclo *for*, de modo a que apenas seja transcrito para o *array tarefasProgramador* uma combinação que seja melhor que a anterior gerada, realizamos uma condição de comparação entre as variáveis *tasksValidas* e *totalTasks*. Cada vez que a *tasksValidas* possuir uma valor maior que a variável *totalTasks*, o *totalTasks* muda o seu valor para o valor da *tasksValidas*. Assim, na próxima iteração do *loop*, quando a função *MelhorComb* for invocada, o valor da variável *totalTasks* será diferente e, dentro da função, a condição *if (counter > totalTasks)* será afetada.

Com isto, quando o ciclo *for* que percorre todas as tarefas for concluído, no vetor *tarefasProgramador* estará a melhor combinação de todas as *tasks* e podemos escrever no ficheiro as tarefas que foram associadas ao programador iterado.

É importante também que as colunas e linhas correspondentes às tarefas que foram escolhidas e atribuídas ao programador passem a ter valor 0 na matriz *MatrizCompativeis*.

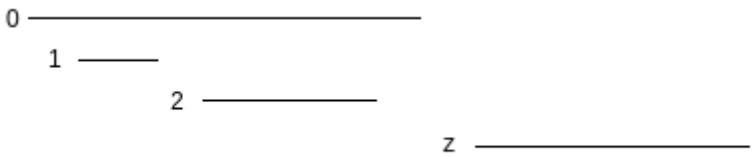
```
    for (int x=0; x<problem->T; x++) {
    if (tarefasProgramador[x] != -1) {
            fprintf(fp,    "Tarefa    que    começa    em    %d    e    acaba    em    %d\n",
problem->task[tarefasProgramador[x]].starting_date,
problem->task[tarefasProgramador[x]].ending_date);
            for (int coluna = 0; coluna<problem->T;coluna++) {
                  MatrizCompativeis[tarefasProgramador[x]][coluna] = 0;
                  MatrizCompativeis[coluna][tarefasProgramador[x]] = 0;
            }
        }
    }
    free(tarefasProgramador);
    free(jaEscolhida);
```

Contudo, da forma em que o código foi elaborado, ele ainda não mostra a melhor combinação de tarefas, não sendo totalmente funcional ainda. Consegue perceber onde está o erro?

Repare que, embora entremos com uma tarefa *z*, o primeiro passo que será realizado ao entrar na função *MelhorComb* é um *loop for* que inicia-se em 0, ou seja, a combinação que será gerada para a tarefa *z* irá seguir os mesmos padrões que a combinação que foi gerada para a tarefa 0. Contudo, entre as próximas duas opções, qual será a melhor?

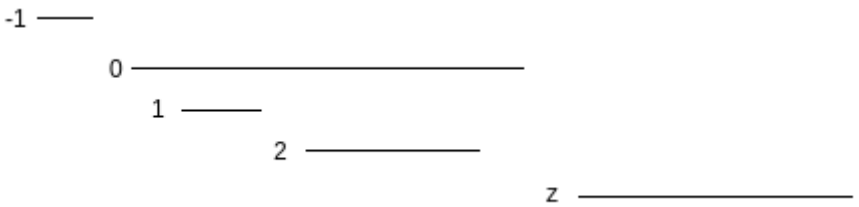
(a) Capturar uma tarefa compatível com *z* com menor data inicial
(b) Capturar uma tarefa compatível com *z* com maior data inicial

Observe a imagem para um melhor entendimento da situação:



Dada a tarefa *z*, se o programador optar por escolher a tarefa 0 (situação (a)), então a combinação será 0 - *z*; porém, se o programador optar por escolher a tarefa 2 (situação (b)), então a combinação será 1 - 2 - *z*.

A forma como as tarefas estão organizadas (ordem crescente de data inicial) colocam-nos na seguinte situação: uma tarefa *x*+1 pode ter uma data final superior ou inferior a uma tarefa *x*, mas nunca terá uma data inicial inferior, pelo que, a probabilidade de haver melhores combinações associadas à tarefa *x*+1 é maior. Dada uma tarefa *y*, se a tarefa *y* é compatível com a tarefa *x* pela data inicial, então também será compatível com a tarefa *x*+1 pela data inicial; mas se a tarefa *y* é compatível com a tarefa *x*+1 pela data inicial, não podemos garantir que seja também compatível com a tarefa *x*.



Note que a tarefa -1 é compatível com a tarefa 0 pela data inicial da tarefa 0 e, consequentemente, também é compatível com a tarefa 2; porém a tarefa 1 é compatível com a tarefa 2 pela data inicial da tarefa 2 e não é compatível com a tarefa 0.

Aplicando esta lógica no nosso código, temos então que dividir um ciclo *for* em dois: o primeiro percorre as tarefas com data inicial superior à tarefa *z* num *loop* de incrementação e o segundo percorre as tarefas com data inicial inferior à tarefa *z* num *loop* de decrementação, tal como foi elaborado na implementação anterior.

```
for (int element=tarefa; element >= 0; element--) {
         if(MatrizCompativeis[tarefa][element] == 1) {
          . . .
         }
         else { continue; }
}
for (int element=tarefa; element >= 0; element--) {
         if(MatrizCompativeis[tarefa][element] == 1) {
         . . .
         }
          else { continue; }
}

if (totalTasks < counter) {
```

```
            for (int x=0; x<problem->T; x++) {
                  tarefasProgramador[x] = -1;
            }
            totalTasks = counter;
            for (int x=0; x<problem->T; x++) {
                  tarefasProgramador[x] = jaEscolhida[x];
            }
}
return counter;
```

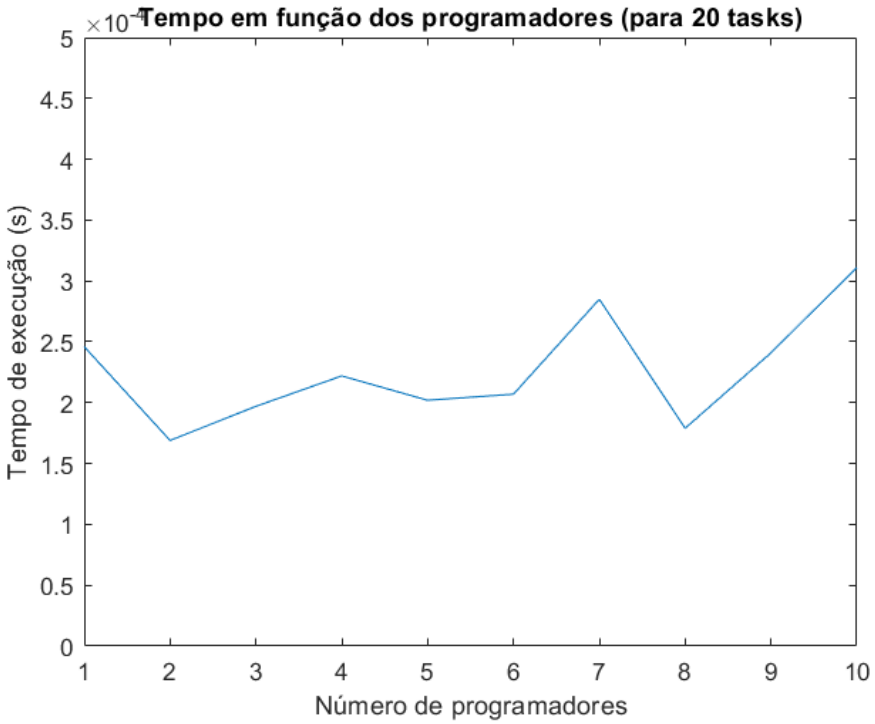


Fig.03 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

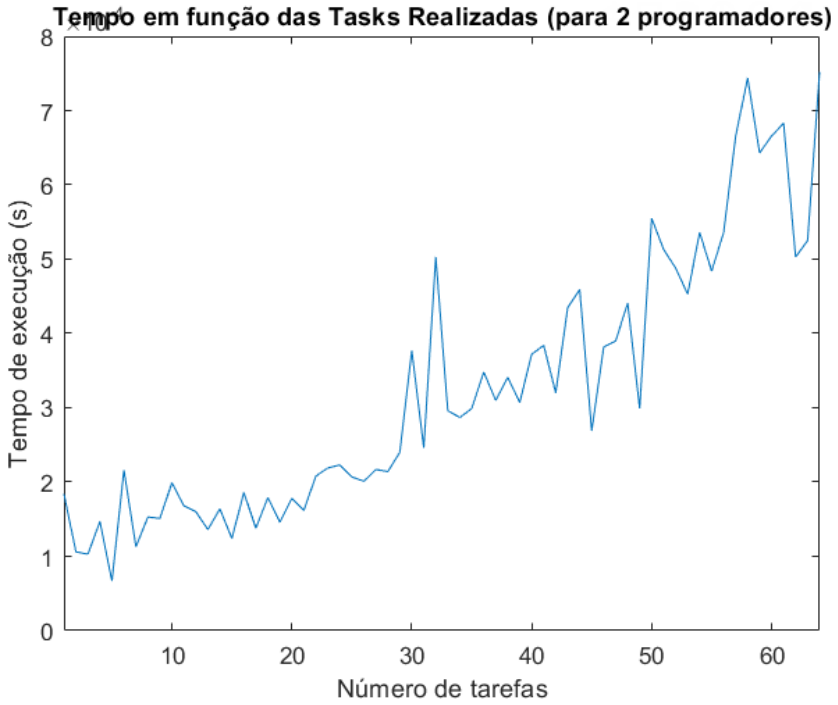


Fig.04 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 64 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

3.4 Quarta Abordagem

Esta abordagem segue a mesma linha de raciocínio utilizada numa fase inicial para encarar o problema, onde a base do raciocínio é: o código será mais eficiente se organizarmos as tarefas pela ordem crescente da data final. Como já foi mencionado anteriormente, no tópico 3.1 Primeira Abordagem, ao organizarmos as *tasks* desta maneira, as tarefas que terminam primeiro ficam numa posição mais perto da inicial e tarefas que possuem uma data inicial baixa mas uma data de término elevada dificilmente serão escolhidas, pois apenas uma passagem pelo *array* já nos daria o máximo número de tarefas que **1** programador é capaz de realizar.

Uma das falhas do código inicial usado na primeira abordagem é causada por deduzirmos que a combinação gerada era obrigatoriamente a melhor combinação, quando isso não se verifica sempre. A combinação gerada numa única passagem por todas as tarefas em busca de uma combinação compatível irá nos fornecer **uma das melhores** combinações, mas não necessariamente a melhor.

Por exemplo, se a combinação fornecida for 3 - 13 - 16 - 19, a maneira correta de interpretar o resultado é: a melhor combinação é constituída por 4 tarefas. De seguida, é fundamental analisar se a combinação mencionada é a melhor ou se existe uma outra combinação melhor.

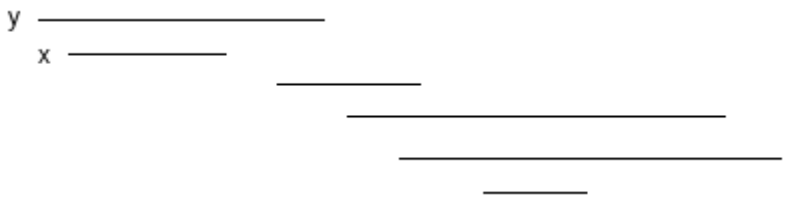
O que faz uma combinação ser melhor do que a outra neste contexto? Ponha em jogo a seguinte situação: existem 6 tarefas (a, b, c, d, f, h) e, ao percorrermos todas as tarefas, concluiu-se que uma das melhores combinações é a - c - d - f, ou seja, não é possível realizar mais do que 4 tarefas. No entanto, a combinação a - b - d - f também é uma combinação

possível, contudo a tarefa b possui uma data final maior do que a tarefa c, pelo que a tarefa c foi escolhida e a b não. Como analisaremos então qual das duas combinações será a melhor opção?

Imagine agora que, ao escolhermos a opção a - c - d - f, deixando livre apenas as tarefas b e h, o programador 2 poderá apenas realizar uma delas porque ambas não são compatíveis. Todavia, se escolhermos a opção a - b - d - f, como a tarefa c tem uma data menor que a tarefa b, já é possível que o programador 2 realize as tarefas c e h. Neste cenário, é mais favorável então escolher a combinação a - b - d - f em vez da combinação gerada inicialmente.

Respondendo agora à pergunta colocada anteriormente, o que faz uma combinação ser melhor do que a outra é ela permitir que, posteriormente, mais tarefas possam ser agregadas. Quando passamos esta condição para linguagem de programação e para este contexto, podemos dizer que é viável substituir a tarefa x pela tarefa y se y é compatível com todas as restantes tarefas da combinação e possui uma duração maior. Isto porque, se y cumprir essas condições, ao escolhermos essa tarefa e deixarmos livre a tarefa x para combinações posteriores, a probabilidade de gerarmos melhores combinações é maior. Note que, se ambas as tarefas y e x são compatíveis com as restantes tarefas escolhidas para a combinação e y tem uma maior duração, isso não significa necessariamente que x e y têm as mesmas tarefas compatíveis.

Para um melhor entendimento, observe o exemplo:



Observe que, ao escolhermos a tarefa y em vez da tarefa x para uma previamente dada combinação, deixamos a tarefa x livre e, consequentemente, o programador seguinte será capaz de realizar 3 tarefas. O mesmo não iria ocorrer caso escolhêssemos a tarefa x para a combinação, pois o programador seguinte iria apenas conseguir realizar 2 tarefas, visto que não há compatibilidade entre a terceira *task* da imagem e a tarefa y.

Explicado o raciocínio, iremos analisar então o código implementado para esta abordagem.

O primeiro passo do nosso código é inicializar duas variáveis, o *array Comb*, onde irá ser armazenada inicialmente uma das melhores combinações e, depois, iremos alterar para que fique guardada a melhor combinação, e a variável *counter* com o valor 0, que servirá para adicionar as tarefas de uma das melhores combinações no vetor *Comb*, de forma a que as tarefas compatíveis fiquem na parte inicial do *array* e a parte final permaneça preenchida com valores -1 - essa atribuição será feita brevemente. Para além disso, colocamos o valor -1 associado às tarefas, simbolizando que ainda não foram atribuídas.

```
int *Comb=(int *) malloc(sizeof(int)*(problem->T));
int counter = 0;
```

```
for (int i=0; i<problem->T; i++) { problem->task[i].assigned_to=-1; }
```

Seguidamente, entramos num ciclo *for* que irá percorrer todos os programadores. Para cada iteração do ciclo, o vetor *Comb* será preenchido com valores iguais a -1, o que indica que aquele programador, até ao momento, não tem uma combinação associada. Posteriormente, entramos num *loop for* que percorrerá todas as tarefas, contudo apenas será do nosso interesse analisar tarefas que ainda não tenham sido atribuídas a um programador, pelo que a primeira condição realizada ao entrarmos no ciclo é, caso a tarefa iterada já tenha um programador associado, passamos para a próxima tarefa do *array*.

```
for (int p=1; p<=problem->P; p++){
for (int i=0; i<problem->T; i++) { Comb[i] = -1; }
        for (int task=0; task<problem->T; task++) {
              if (problem->task[task].assigned_to!= -1) { continue; }
        . . .
}
```

Caso a tarefa iterada ainda não tenha sido atribuída a nenhum programador, iremos analisar se é possível atribuí-la; se o programador ainda não estiver ocupado (o valor associado ao seu parâmetro *busy* é -1) ou se estiver ocupado mas a data final da tarefa que se encontra em realizar for menor que a data inicial da tarefa iterada, então iremos atribuir a tarefa iterada ao programador e adicionamo-la no *array Comb*, incrementando de seguida a variável *counter*. Após todas as tarefas terem sido percorridas e lembrando que elas estão organizadas por ordem crescente da data final, a combinação presente no vetor *Comb* será uma das melhores combinações possíveis.

```
if (problem->busy[p] == -1) {
        problem->task[task].assigned_to=p;
        problem->busy[p]=problem->task[task].ending_date;
        Comb[counter]=task;
        counter++;
}
else {
        if (problem->busy[p]<problem->task[task].starting_date) {
              problem->task[task].assigned_to=p;
              problem->busy[p]=problem->task[task].ending_date;
              Comb[counter]=task;
              counter++;
        }
}
```

Nesta etapa, iremos então verificar se é possível, na combinação fornecida, substituir tarefas de forma a deixar livre tarefas que posteriormente poderão gerar melhores combinações para o dado problema. Para tal, inicializamos dois ciclos *for*; o segundo percorrerá todas as tarefas do problema apresentado e o primeiro irá percorrer todos os elementos do vetor *Comb* -

como o vetor possui elementos com valor -1 depois da posição *counter*-1, apenas é relevante que este ciclo *for* itere até *counter*, exclusive. Apenas queremos analisar *tasks* que ainda não tenham sido designadas, pelo que a primeira condição que encontramos filtra exatamente isso.

```
    for (int task=0; task<counter; task++) {
    int         duration1         =         problem->task[Comb[task]].ending_date         -
problem->task[Comb[task]].starting_date;
            for (int element=0; element<problem->T; element++) {
    int          duration2          =          problem->task[element].ending_date         -
problem->task[element].starting_date;
                  if (problem->task[element].assigned_to ==-1) {
                  . . .
    }
```

Se a tarefa não tiver sido atribuída a nenhum programador, irá passar pela filtragem de conteúdo. Essa filtragem divide-se em três condições: se estamos a procurar tarefas que possam substituir o primeiro elemento do *array Comb*, ou seja, se *task* tiver o valor 0, será apenas necessário verificar se existe alguma tarefa compatível com o próximo elemento do *array* (estão organizados por data final crescente) mas com uma duração superior à duração associada ao primeiro elemento do *Comb*; se estamos a procurar tarefas que possam substituir o último elemento do *array Comb*, ou seja, se a *task* tiver valor *counter*-1, será apenas necessário verificar se existe alguma tarefa compatível com o anterior elemento do *array* mas com uma duração superior à duração associada ao último elemento do *Comb*; se estamos a procurar tarefas que possam substituir elementos que não se encontram nas extremidades do *array Comb*, então precisamos de verificar se a tarefa analisada é compatível com o elemento anterior e o com o elemento posterior e, ainda, se possui uma duração superior à duração associada ao elemento do *Comb* que queremos substituir.

Caso o elemento analisado passe na filtragem, atribuímos essa tarefa ao programador, removemos do *array Comb* a tarefa que estava antes associada e alteramos o seu valor de atribuição para -1.

```
    if (task == 0) {
            if ((problem->task[element].ending_date <
            problem->task[Comb[task+1]].starting_date) && (duration2>duration1)){
                  problem->task[Comb[task]].assigned_to=-1;
                  problem->task[element].assigned_to= p;
                  Comb[task] = element;
            }
    }
    else if (task== (counter -1)) {
            if ((problem->task[element].starting_date >
            problem->task[Comb[task-1]].ending_date) && (duration2>duration1)){
                  problem->task[Comb[task]].assigned_to=-1;
                  problem->task[element].assigned_to= p;
                  Comb[task] = element;
```

```
        }
}
else {
        if ((problem->task[element].starting_date >
 problem->task[Comb[task-1]].ending_date) && (problem->task[element].ending_date <
 problem->task[Comb[task+1]].starting_date) && (duration2>duration1)){
                problem->task[Comb[task]].assigned_to=-1;
                problem->task[element].assigned_to= p;
                Comb[task] = element;
        }
}
```

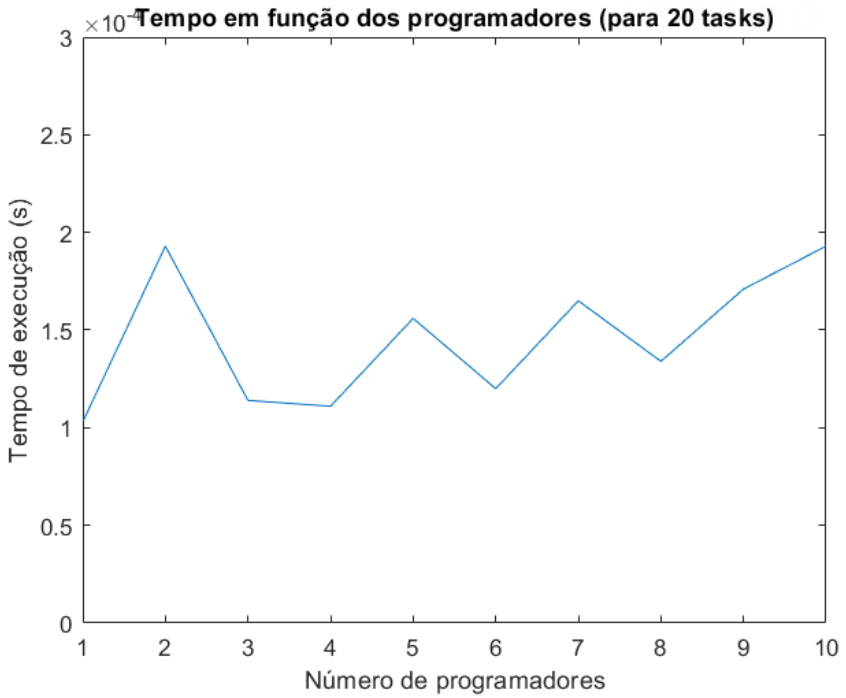


Fig.05 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

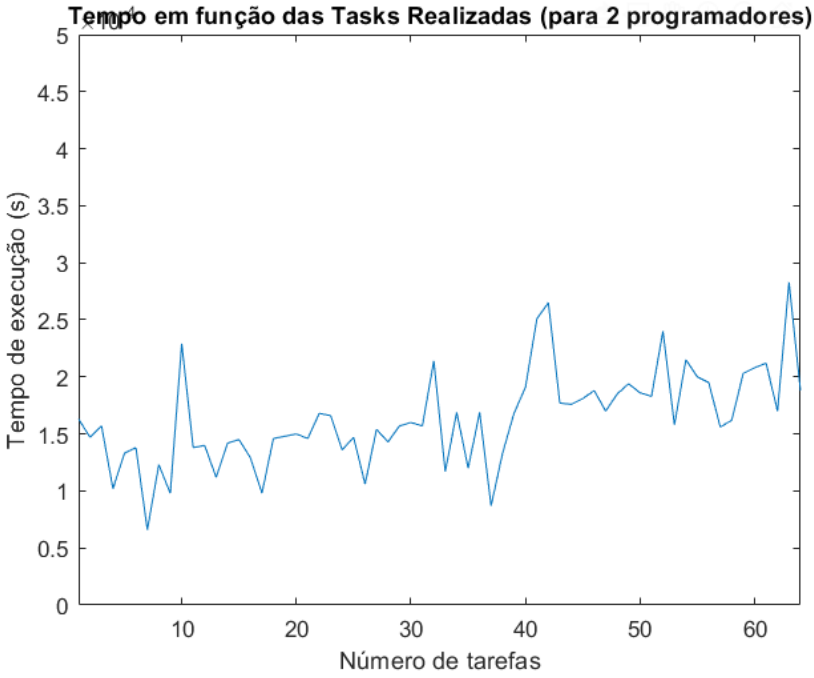


Fig.06 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 64 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

3.5 Quinta Abordagem

Com a realização de várias implementações chegámos à conclusão de que as implementações anteriormente descritas não apresentavam sempre os melhores resultados porque não testavam todas as combinações possíveis. Por isso, nesta implementação, usamos uma abordagem baseada na geração de combinações binárias, que corrigirá essa falha, testando **todas** as combinações possíveis.

Dado um número de tarefas, *problem->T*, serão geradas 2^T combinações binárias de comprimento *problem->T*. Exemplo: Se existirem 3 tarefas, as combinações geradas serão 000,001,010,011,100,101,110 e 111. O 1 nestas combinações significa que a tarefa em questão terá que ser realizada, e o 0 significa que a tarefa não será realizada. Pretende-se assim então testar, para cada tarefa, se a combinação é possível tendo em conta o número de programadores disponíveis e as datas de início e fim de cada tarefa.

Para gerar estas combinações fazemos uso da função *gerarCombinacoes*, que tem como argumentos de entrada um int n que vai conter o tamanho da combinação a gerar (neste caso *problem->T*), um array de *int*'s com n espaços (*int arr[ ]* que irá guardar uma combinação gerada), um *int i* (que vai funcionar como índice) e um *array de arrays de int*'s (*int **combinações* que irá guardar todas as combinações geradas). A função consiste então em gerar todas as combinações binárias de tamanho *n* de forma recursiva onde será gerada primeiro uma combinação toda a 0's que será incrementada de forma binária até que seja gerada uma combinação toda a 1's.

```
void gerarCombinacoes(int n, int arr[], int i,int **combinacoes) {
   if (i == n) {
       addToArray(arr, n,combinacoes);
       return;
   }
   arr[i] = 0; gerarCombinacoes(n, arr, i + 1,combinacoes);
   arr[i] = 1; gerarCombinacoes(n, arr, i + 1,combinacoes);
}
```

Cada combinação será guardada no *array arr*, e por cada combinação completa será chamada a função *addToArray* que irá adicionar a combinação ao *array combinacoes*. A variável *nrComb* funciona como contador de todas as combinações adicionadas, sendo usada como índice para o armazenamento da combinação no *array combinacoes*.

```
static int nrComb=0;
void addToArray(int arr[], int n, int**combinacoes) {
   for (int i = 0; i < n; i++) {
     combinacoes[nrComb][i]=arr[i];
   }
   nrComb++;
}
```

Agora, na função *solve*, começamos por inicializar as estruturas que nos serão úteis ao longo da implementação. O *array melhorAssignedTo*, com *problem->T* espaços, guarda a melhor combinação de atribuição de tarefas. O *array de arrays combinacoes* tem 2^*problem->T* arrays, cada um com *problem->T*espaços. Este irá guardar todas as combinações binárias possíveis, sendo que cada *array* é uma combinação. O *array de arrays tarefasProgramador* tem *problem->P* arrays (um para cada programador), cada um com *problem->T* espaços, guarda as tarefas que cada programador faz numa dada combinação.

Antes da resolução do problema, inicializamos os vetores *combinacoes* e *melhorAssignedTo* a -1.

```
combinacoes=(int**)malloc(sizeof(int*)*pow(2,problem->T));
 for(int i=0;i<pow(2,problem->T);i++)
 {
   combinacoes[i]=(int*)malloc(sizeof(int)*problem->T);
 }

 tarefasProgramador=(int**)malloc(sizeof(int*)*problem->P);
 for(int i=0;i<problem->P;i++)
 {
   tarefasProgramador[i]=(int*)malloc(sizeof(int)*problem->T);
 }

 melhorAssignedTo=(int*)malloc(sizeof(int)*problem->T);

 int n = problem->T;
 int arr[n];
```

```
for(int i=0;i<pow(2,problem->T);i++)
 {
   for(int k=0;k<problem->T;k++)
   {
     combinacoes[i][k]=-1;
   }
 }
  gerarCombinacoes(n, arr, 0,combinacoes);

 for(int i=0;i<problem->T;i++)
 {
   melhorAssignedTo[i]=-1;
 }
```

Esta implementação tem duas opções: a opção que contabiliza os lucros e a opção que ignora os lucros. Nesta secção iremos focar-nos apenas na opção que ignora os lucros.

O objetivo é então, encontrar a combinação que nos dê o maior número de tarefas realizadas possível para um determinado número de programadores.

Para contabilizar o número de tarefas realizadas usaremos a variável *nrTasksGeral*, inicializada a 0, que guardará o valor máximo de tarefas realizadas até ao momento.

Agora iteramos por todas as combinações e inicializamos os vetores *problem->busy*, *tarefasProgramador* e *assigned_to* todos a -1. A variável *nrTasks*, que armazena o número de tarefas realizadas para cada combinação é também inicializada a 0 para cada combinação.

```
nrTasksGeral=0;
   for(int comb=0;comb<pow(2,problem->T);comb++){
     for(int i=0;i<problem->P;i++){
       problem->busy[i]=-1;
     }
     for(int i=0;i<problem->P;i++) {
       for(int k=0;k<problem->T;k++){
         tarefasProgramador[i][k]=-1;
       }
     }
     for(int i=0;i<problem->T;i++) {
       problem->task[i].assigned_to=-1;
     }
     nrTasks=0;
```

Iteramos agora por todos os programadores e, para cada um, iteramos por todas as tarefas.

Caso a tarefa *tar* da combinação *comb* tenha de ser feita, verificamos se o programador *prog* está disponível. Caso esteja e a tarefa *tar* não esteja atribuída, atribui-se a tarefa *tar* ao programador *prog*, guardando o valor de *tar* em *tarefasProgramador*. O valor de *busy* do programador *prog* é definido com o valor da *ending_date* da tarefa *tar*, define-se que a tarefa *tar*

está atribuída ao programador *prog* e incrementa-se o número de tarefas realizadas na combinação atual.

```
for(int prog=0;prog<problem->P;prog++)
     {
       for(int tar=0;tar<problem->T;tar++)
       {
         if(combinacoes[comb][tar]==1)
         {
           if(problem->busy[prog]==-1)
           {
             if(problem->task[tar].assigned_to==-1)
             {
               tarefasProgramador[prog][nrTasks]=tar;
                 problem->busy[prog]=problem->task[tar].ending_date;
               problem->task[tar].assigned_to=prog;
               nrTasks++;
             }
           }
```

Caso o programador não esteja disponível e a tarefa não esteja atribuída, vamos comparar as *starting_dates* com as *ending_dates*. Se a *starting_date* da tarefa for maior que o valor de *busy* (*ending_date* da anterior) então o programador pode realizá-la porque não vai haver sobreposição. Se não houver sobreposição atribui-se então a tarefa *tar* ao programador *prog*, realizando as mesmas operações previamente descritas.

```
else {
     if(problem->task[tar].assigned_to==-1)  {
         if(problem->busy[prog]<problem->task[tar].starting_date) {
                 tarefasProgramador[prog][nrTasks]=tar;
                 problem->busy[prog]=problem->task[tar].ending_date;
                 problem->task[tar].assigned_to=prog;
                 nrTasks++;
         }
     }
```

No final das iterações por todos os programadores e por todas as tarefas, vamos definir a variável *flagE* a zero.

A variável *flagE* vai servir como deteção de inviabilidades. É definida a zero aqui e vai passar por algumas condições. Caso o seu valor se mantenha a zero quer dizer que a combinação é viável. Caso o seu valor se altere para 1 quer dizer que a combinação não reúne as condições necessárias para ser viável. Para ser viável, todas as tarefas que na combinação tenham o valor '1' têm que ser atribuídas. Caso haja tarefas que tenham o valor '1' na combinação mas que não tenham sido atribuídas, quer dizer que não há programadores suficientes para as realizar a todas, então a combinação revela-se inviável.

```
int flagE=0;
```

A verificação passa por iterarmos por cada elemento da combinação e verificar se a tarefa tem que ser realizada (*combinacoes[comb][i]*==1) e verificar se essa tarefa não foi atribuída. Caso essas duas condições se verifiquem, a combinação é inviável (a variável *flagE* é definida a 1).

```
for(int i=0;i<problem->T;i++)  {
       if(combinacoes[comb][i]==1) {
         if(problem->task[i].assigned_to==-1) {
           flagE=1; break;
         }
       }
     }
```

Caso todas as tarefas que tinham que ter feitas tiverem sido feitas (*flagE*==0), vamos verificar se a combinação realizada nesta iteração é melhor do que a melhor combinação até agora.

Para isso comparamos a variável *nrTasks* e *nrTasksGeral*. Se o número de tarefas realizadas nesta combinação for superior ao número de tarefas realizadas geral, vamos reinicializar o vetor *melhorAssignedTo* a -1 e vamos atualizar o valor de *nrTasksGeral* para que contenha o melhor valor atual (*nrTasks*). De seguida, caso o elemento *tarefasProgramador[i][k]* seja uma tarefa (o seu valor seja diferente de -1), atribuímos ao *melhorAssignedTo*, na posição correspondente à tarefa *tarefasProgramador[i][k]* o valor de *i* (do programador).

```
if(flagE==0){
       if(nrTasks>nrTasksGeral)  {
         for(int i=0;i<problem->T;i++) {
           melhorAssignedTo[i]=-1;
         }
         nrTasksGeral=nrTasks;
         for(int i=0;i<problem->P;i++) {
           for(int k=0;k<problem->T;k++) {
             if(tarefasProgramador[i][k]!=-1)  {
               melhorAssignedTo[tarefasProgramador[i][k]]=i;
             }
           }
         }
       }
     }
```

No final de todas as combinações, copiamos o valor do *array melhorAssignedTo* para o *array assignedTo* do problema e imprimimos os dados no ficheiro. Os dados impressos no ficheiro estarão ordenados por programador e, para cada programador, serão impressas as respetivas tarefas atribuídas.

```
for(int i=0;i<problem->T;i++)  {
```

```
          problem->task[i].assigned_to=melhorAssignedTo[i];
        }
        fprintf(fp,"Solução sem Lucros\n\n");
        for(int p=0;p<problem->P;p++)  {
          fprintf(fp,"\nPara o Programador %d\n",(p+1));
          for(int t=0;t<problem->T;t++)  {
            if(problem->task[t].assigned_to==p)  {
              fprintf(fp,"Foi atribuída a task que começa em %d e acaba em
%d\n",problem->task[t].starting_date,problem->task[t].ending_date);
            }
          }
        }
        fprintf(fp,"\nForam feitas %d tarefas.\n",nrTasksGeral);
        fprintf(fp,"-------------------------------------\n");
        }
```

Apesar desta implementação se revelar correta em termos de resultados, ela não é completamente funcional visto que se correrrmos o processo para mais de 24 tarefas o terminal mata o processo. Isto deve-se ao facto de os resultados de todas as combinações serem armazenados numa estrutura de dados, o que acaba por sobrecarregar a memória utilizada pelo programa. Para corrigir este problema e de forma a fazer o programa funcional para um maior número de tarefas fizemos ligeiras alterações no funcionamento do programa, as quais serão descritas na secção 3.6.

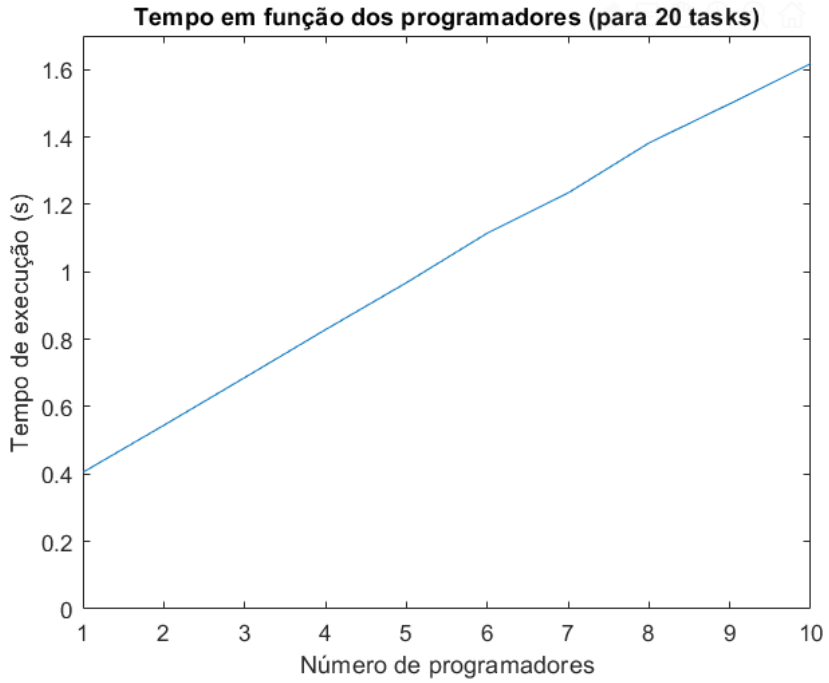


Fig.07 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(1) segundos]

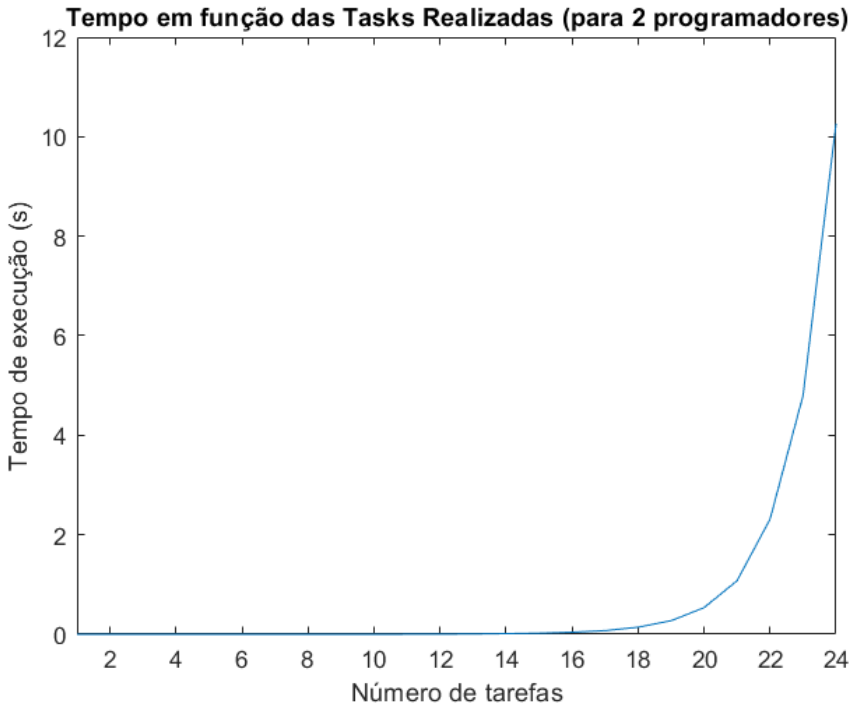


Fig.06 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 24 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(1) segundos]

3.6 Sexta Abordagem

Por último, e como a melhor abordagem para a resolução, temos a nossa sexta implementação do problema que não é nada mais nada menos que uma melhoria da quinta abordagem. Para além da optimização de ciclos *for* para inicialização de variáveis, também chegamos à conclusão que não seria necessário armazenar as combinações realizadas numa matriz, pois isso seria alocar espaço de forma desnecessária e, inclusive, não estava a mostrar-se ser possível de realizar, pelo que poderíamos apenas analisar cada combinação quando a mesma fosse realizada sem a guardar.

Para além disso, chegamos à conclusão de que a implementação utilizada funciona para as duas situações: quando pretendemos ignorar os lucros e quando pretendemos contabilizar os lucros. Isto porque, quando nos referimos a ignorar os lucros, estamos a dizer que cada tarefa tem um *profit* igual a 1. Ora, se todas as tarefas possuem um mesmo lucro e pretendemos realizar o máximo número de tarefas, então estamos à procura da combinação que nos permita obter o maior lucro possível.

Assim sendo, a explicação detalhada do código será realizada na secção 4, uma vez o código utilizado para a sexta abordagem para um *input* de I igual a 1 e o código usado para a segunda abordagem de um *input* de I igual a 0 é semelhante, mudando apenas o que é impresso no ficheiro - neste caso, o lucro não é impresso.

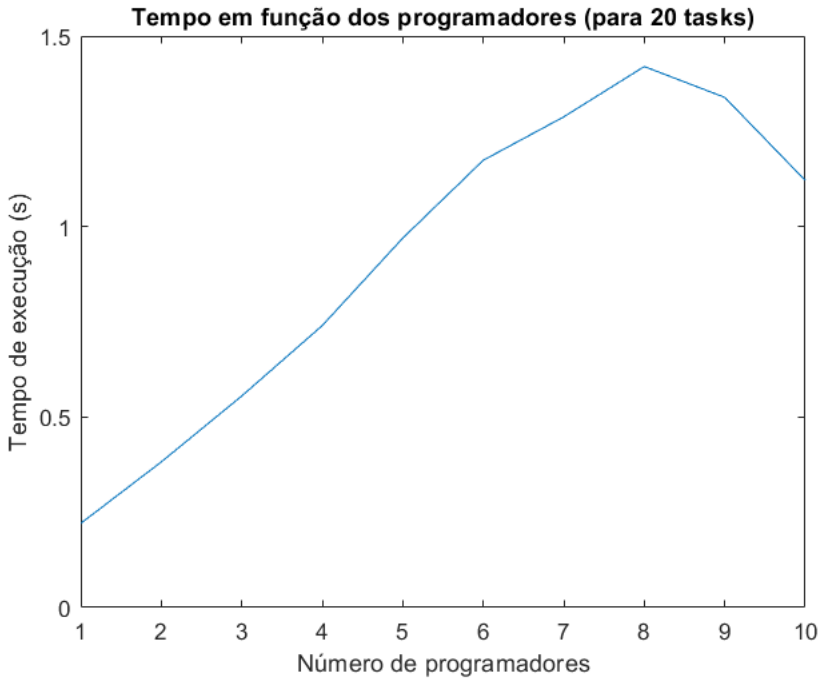


Fig.09 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

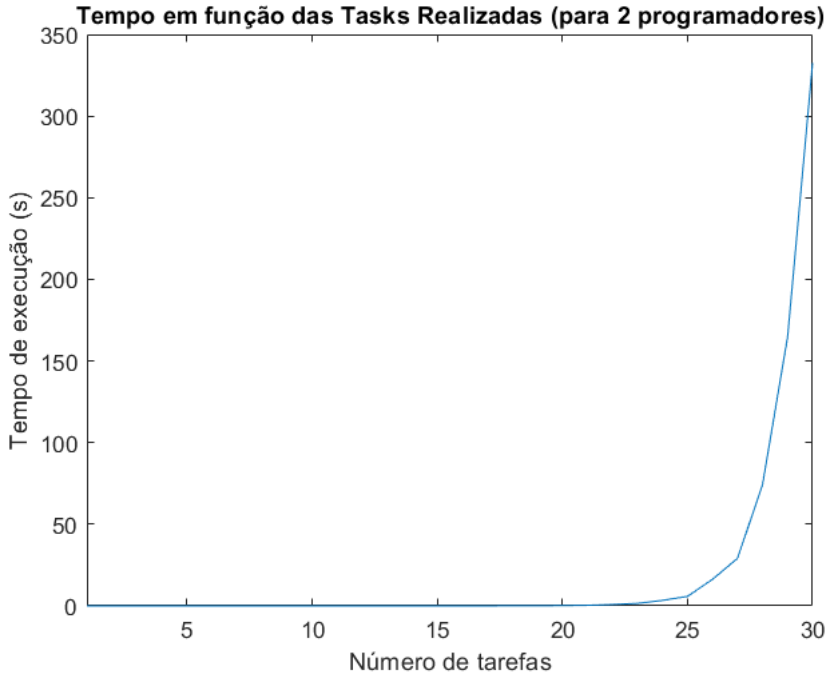
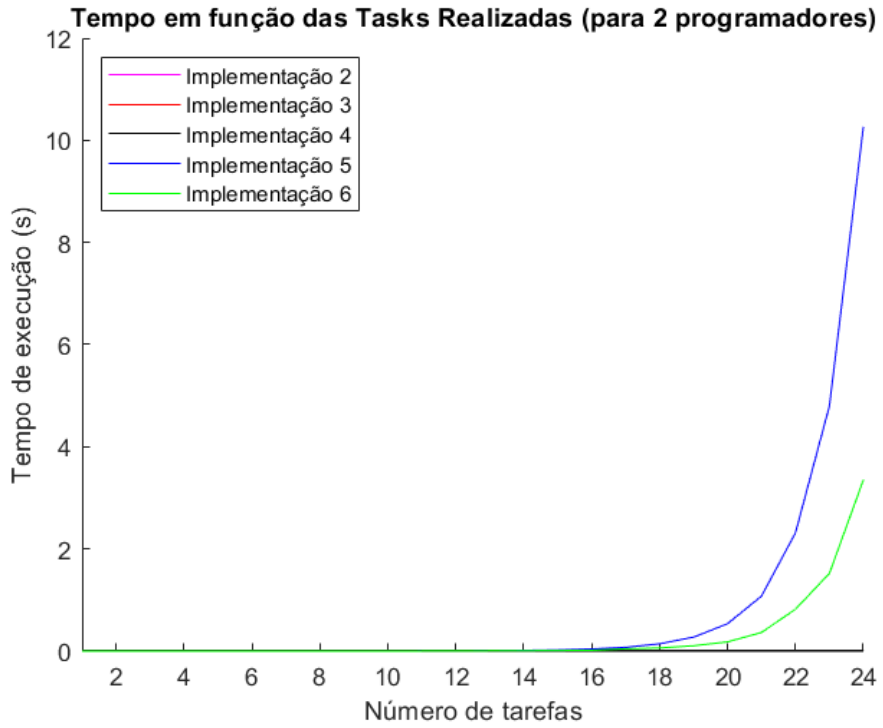


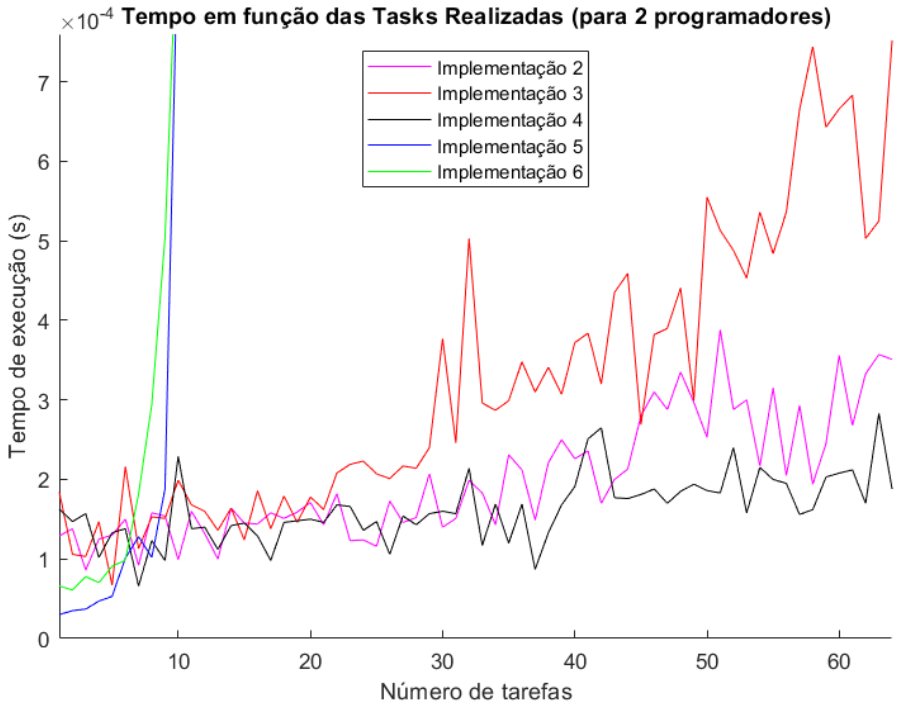
Fig.10 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 30 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(-4) segundos]

3.7 Comportamento de cada abordagem em relação ao tempo

- **Para um número fixo de programadores**



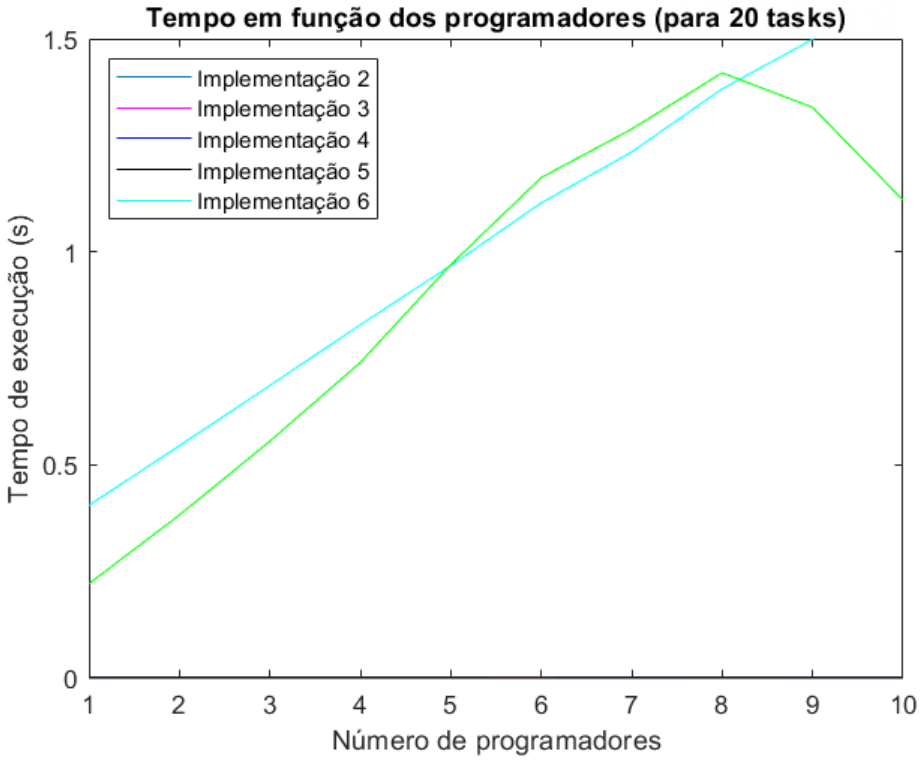
Visto que os tempos de execução apresentam uma grande discrepância, tornando-se impossível a visualização das implementações 2, 3 e 4, observe o gráfico seguinte com um eixo y limitado para um número pequeno.



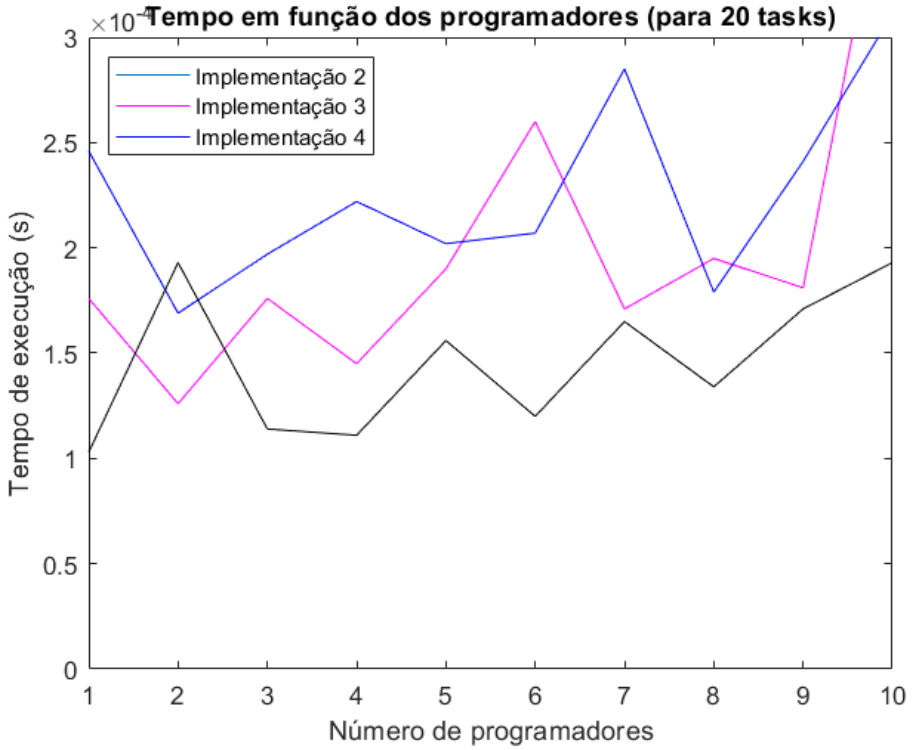
Como é possível observar no gráfico de comparação dos tempos de execução entre as abordagens anteriormente explicadas, para um máximo de tarefas igual a 24 (uma vez que a

abordagem 1 interrompe o processo quando o número de *tasks* é superior a 24) e um número de programadores constante igual a 2, tendo em conta que as abordagens 2, 3 e 4 não apresentam sempre o resultado correto - uma vez que não geram todas as combinações, como já foi explicado previamente -, a segunda implementação mostra-se mais eficiente que a primeira implementação. Isto ocorre porque as combinações geradas na segunda implementação não são guardadas em nenhuma estrutura e há uma validação da viabilidade da combinação. Num cenário hipotético, onde as primeiras três implementações apresentassem um desempenho completamente funcional, a melhor abordagem seria a abordagem 4, uma vez que o seu tempo de execução é o menor dentro dos 5 gráficos.

- **Para um número fixo de tarefas**



Visto que os tempos de execução apresentam uma grande discrepância, tornando-se impossível a visualização das implementações 2, 3 e 4, observe o gráfico seguinte com um eixo y limitado para um número pequeno.



Como é possível observar no gráfico de comparação dos tempos de execução entre as abordagens anteriormente explicadas, para um máximo de programadores igual a 10 e um número de tarefas constante igual a 20, tendo em conta que as abordagens 2, 3 e 4 não apresentam sempre o resultado correto - uma vez que não geram todas as combinações, como já foi explicado previamente -, a segunda implementação mostra-se mais eficiente que a primeira implementação. Isto ocorre porque as combinações geradas na segunda implementação não são guardadas em nenhuma estrutura e há uma validação da viabilidade da combinação. Num cenário hipotético, onde as primeiras três implementações apresentassem um desempenho completamente funcional, a melhor abordagem seria a abordagem 4, uma vez que o seu tempo de execução é o menor dentro dos 5 gráficos.

- **Para os exemplos dos slides**

Observe o tempo de execução para a abordagem 6, que se revelou a melhor abordagem, para os exemplos demonstrados nos slides.

Nº de programadores	Nº de tarefas	Total de tarefas	Lucro total (não valorizando o lucro)	Tempo de execução
4	20	12	12	7.092e-01
6	30	19	19	1.389e+03

-

4. Exposição das abordagens (valorizando o lucro)

Neste tópico iremos apenas focar nas abordagens utilizadas para resolver o enunciado referente à procura da melhor combinação de forma a que os programados realizem o máximo lucro possível.

Cada abordagem será acompanhada de dois gráficos que traduzem o tempo de execução do *script* em função do número de tarefas para um número de programadores fixo (2 programadores) e o tempo de execução do *script* em função do número de programadores para um número de tarefas fixo (20 tarefas). Esses gráficos serão analisados num só no último tópico desta secção.

4.1 Primeira Abordagem

Para a implementação com lucros, o raciocínio é muito parecido com o da implementação sem lucros 3.5. A diferença está na existência de duas variáveis adicionais: *profitAtual* e *profitGeral*. A variável *profitAtual* guarda o lucro obtido na combinação atual, enquanto que a variável *profitGeral* guarda o melhor lucro obtido até ao momento. Esta última é definida a 0 antes da iteração pelas combinações, e a variável *profitAtual* é inicializada a 0 para cada combinação, tal como a variável *nrTasks*.

A diferença principal desta implementação para a implementação sem lucros é que, sempre que uma tarefa é adicionada, à variável *profitAtual* é somado o lucro da tarefa em questão.

```
profitAtual = profitAtual + problem->task[tar].profit;
```

Para cada combinação, tal como na implementação sem lucros, vamos usar a variável flagE para verificarmos se a combinação é viável ou não. Caso seja viável, vamos comparar os lucros em vez de compararmos o número de tarefas. Sendo assim verificamos se a variável *profitAtual* é maior que a variável *profitGeral*. Caso seja, repetimos o mesmo processo da outra implementação, sendo a única diferença o facto de também atualizarmos o valor de *profitGeral* com o valor do melhor lucro até ao momento (armazenado na variável *profitAtual*).

```
if(flagE==0) {
       if(profitAtual>profitGeral) {
         for(int i=0;i<problem->T;i++) {
           melhorAssignedTo[i]=-1;
         }
         nrTasksGeral=nrTasks;
         profitGeral=profitAtual;
         for(int i=0;i<problem->P;i++){
           for(int k=0;k<problem->T;k++){
             if(tarefasProgramador[i][k]!=-1) {
               melhorAssignedTo[tarefasProgramador[i][k]]=i;
```

```
                }
            }
        }
    }
}
```

De seguida, tudo ocorre da mesma forma. No final atualizamos o valor de *problem->total_profit* com o valor de *profitGeral* e imprimimos os resultados tal como fizemos na implementação 3.5, com a diferença de que desta vez imprimimos também o lucro referente a cada tarefa realizada e o lucro total.

```
problem->total_profit=profitGeral;
```

Tal como na implementação sem lucros (3.5), esta implementação não é completamente funcional visto que se correrrmos o processo para mais de 24 tarefas o terminal mata o processo. Tal como na implementação 3.5, isto deve-se ao facto de os resultados de todas as combinações serem armazenados numa estrutura de dados, o que acaba por sobrecarregar a memória utilizada pelo programa. Para corrigir este problema e de forma a fazer o programa funcional para um maior número de tarefas fizemos ligeiras alterações no funcionamento do programa, as quais serão descritas na secção 4.2.

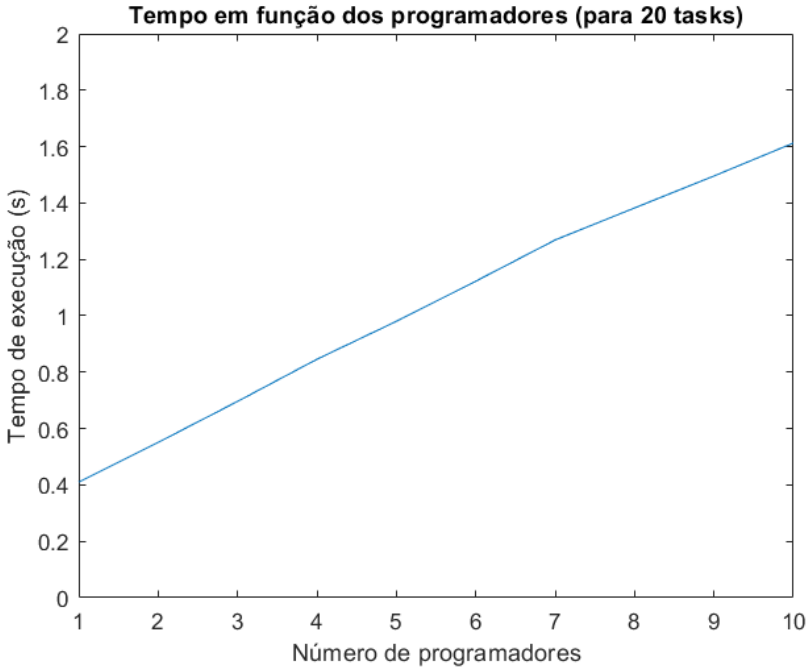


Fig.11 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(1) segundos]

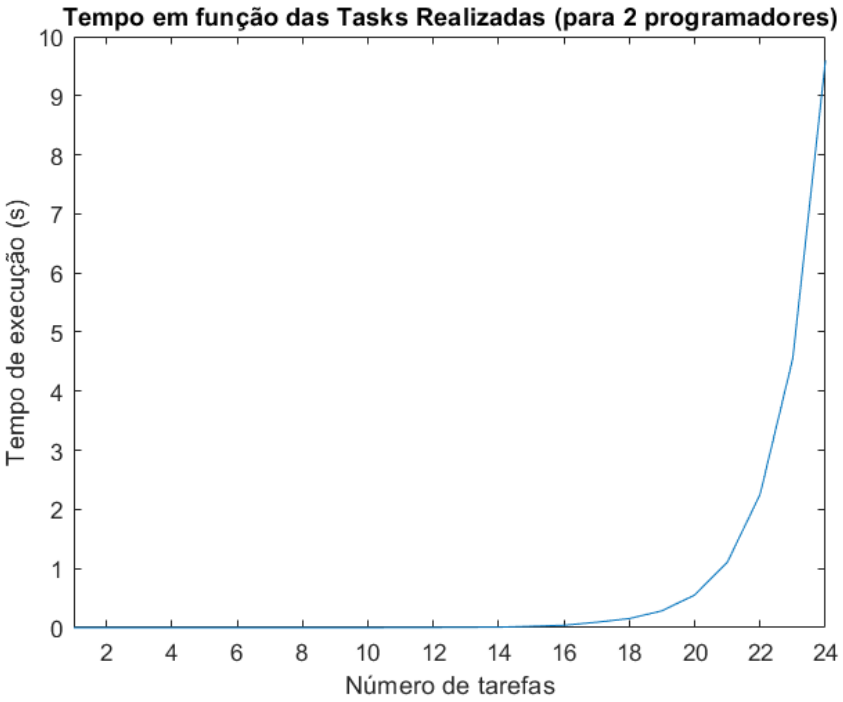


Fig.12 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 24 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(1) segundos]

4.2 Segunda Abordagem

Esta abordagem consiste na melhoria das abordagens cinco (sem lucros) e um (com lucro) e na unificação das duas, pois, como já foi mencionado anteriormente, resolver o problema para que os programadores consigam realizar o maior número de tarefas possível não é nada mais, nada menos, que resolver o problema de forma a obter o melhor lucro possível quando as tarefas possuem todas o mesmo lucro - no caso, todas possuem um lucro igual a 1.

Assim como as outras duas abordagens mencionadas, teremos a inicialização de um *array* *melhorAssignedTo* e de uma matriz *tarefasProgramador*, assim como as variáveis *n* e *arr[n]*.

```
int *melhorAssignedTo; int **tarefasProgramador;
tarefasProgramador=(int**)malloc(sizeof(int*)*problem->P);
for(int i=0;i<problem->P;i++){
    tarefasProgramador[i]=(int*)malloc(sizeof(int)*problem->T);
}
melhorAssignedTo=(int*)malloc(sizeof(int)*problem->T);
for(int i=0;i<problem->T;i++) { melhorAssignedTo[i]=-1; }
int n = problem->T; int arr[n];
```

De seguida, iremos gerar as combinações possíveis através da função *generateAllBinaryStrings* com um funcionamento semelhante à função *gerarCombinações*, mencionada anteriormente, porém com a diferença de que as combinações não serão adicionadas a nenhum *array* mas sim avaliadas na própria hora. Inicialmente, colocámos uma condição *if* que avaliava a viabilidade no seguinte sentido: se eu possuo 5 programadores e pretendo saber a melhor combinação para 2 tarefas, a execução de um programa para tal é desnecessária, pois as duas tarefas serão atribuídas a dois dos cinco programadores e três deles ficarão sem qualquer tarefa. No entanto, para construção dos gráficos e como essa instrução não foi esclarecida em nenhum lado do enunciado, optamos por retirar essa condição *if*, o que afeta o tempo de execução do programa, tornando-o mais lento. (Exemplo: para 30 tarefas e 6 programadores, todas as combinações que exigiam a execução de menos de 6 tarefas não seriam realizadas, logo, por exemplo, mais de 300 combinações não seriam realizadas. Isso iria melhorar muito o tempo de execução do programa, porém o *input* 2020 2 5 0 é um *input* válido, pelo que não passaria nessa condição e, consequentemente, o código não seria executado.) Posteriormente, concluímos que podíamos fazer uma filtragem dessa condição *if*: se o *problem->T* fosse menor que o *problem->P*, essa condição *if* não seria executada, caso contrário, seria executada de forma a melhorar o tempo de execução.

```
    void generateAllBinaryStrings(int n, int arr[], int i, int **tarefasProgramador, int
*melhorAssignedTo, problem_t *problem){
      if (i == n) {
            if (problem->T < problem->P) {
                  function(arr, tarefasProgramador, problem, melhorAssignedTo);
            }
            else {
```

```
                    int contadorUm = 0;
                    for (int a=0; a<problem->T; a++) {
                          if (arr[a] == 1) {
                                contadorUm++;
                          }
                    }
                    if (contadorUm >= problem->P) {
                           function(arr, tarefasProgramador, problem, melhorAssignedTo);
//quando já fez a combinação ele corre a função
                    }
               }
              return;
       }
       arr[i] = 0;
       generateAllBinaryStrings(n, arr, i + 1, tarefasProgramador, melhorAssignedTo,
problem);
       arr[i] = 1;
       generateAllBinaryStrings(n, arr, i + 1, tarefasProgramador, melhorAssignedTo,
problem);
    }
```

Seguidamente, corremos a função *function* cujo código é semelhante a parte do código explicado no sub-tópico 4.1 porém com algumas melhorias, não carecendo assim de qualquer explicação escrita.

```
    void function(int *comb, int **tarefasProgramador, problem_t *problem, int
*melhorAssignedTo) {
          static int nrTasksGeral=0; int profitAtual = 0; static int profitGeral=0; int
nrTasks=0; int stop = 0;
       for(int i=0;i<problem->P;i++){ //Inicializar o vetor tarefasProgramador a -1
              problem->busy[i]=-1;
             for(int k=0;k<problem->T;k++){
                    tarefasProgramador[i][k]=-1;
                    if (stop < problem->T) {
                          problem->task[k].assigned_to=-1;
                          stop++;
                    }
             }
        }
        for(int prog=0;prog<problem->P;prog++) {//Para cada programador
             for(int tar=0;tar<problem->T;tar++) {
                   if(comb[tar]==1) {//Caso a tarefa tar da combinacao comb possa ser
feita
                         if(problem->busy[prog]==-1) { //Caso o programador esteja
disponível
                              if(problem->task[tar].assigned_to==-1) {
                                    tarefasProgramador[prog][nrTasks]=tar;
                                    problem->busy[prog]=problem->task[tar].ending_date;
                                    problem->task[tar].assigned_to=prog;
                                    nrTasks++;
                                    profitAtual=profitAtual+problem->task[tar].profit;
```

```
                        }
                         }
                         else { //Caso o programador esteja ocupado
                               if(problem->task[tar].assigned_to==-1){

if(problem->busy[prog]<problem->task[tar].starting_date){
                                     tarefasProgramador[prog][nrTasks]=tar;

problem->busy[prog]=problem->task[tar].ending_date;
                                     problem->task[tar].assigned_to=prog;
                                     nrTasks++;

profitAtual=profitAtual+problem->task[tar].profit;
                                  }
                               }
                         }
                   }
             }
          }
       int flagE=0;
       for(int i=0;i<problem->T;i++) {//Para cada elemento da combinacao
             if(comb[i]==1) {//Caso a task tenha que ser realizada
                   if(problem->task[i].assigned_to==-1) {//Caso a task não tenha sido
atribuída
                         flagE=1;
                         break;
                   }
             }
       }

         if(flagE==0){ //Caso todas as tasks que tinham que ser feitas tiverem sido feitas
              if(profitAtual>profitGeral) {
                   for(int i=0;i<problem->T;i++) { //Inicializar o vetor melhorAssignedTo
a -1
                         melhorAssignedTo[i]=-1;
                   }
                   nrTasksGeral=nrTasks;
                   profitGeral=profitAtual;
                   for(int i=0;i<problem->P;i++) {
                         for(int k=0;k<problem->T;k++) {
                               if(tarefasProgramador[i][k]!=-1) { //Caso o elemento seja
uma tarefa
                                     melhorAssignedTo[tarefasProgramador[i][k]]=i;
                               }
                         }
                   }
             }
       }
       problem->total_profit=profitGeral;
    }
```

Por último, após gerar todas as combinações possíveis e analisá-las conforme a sua viabilidade, realizamos a secção do código designada à impressão da melhor combinação possível no ficheiro.

```
    for(int i=0;i<problem->T;i++) {
        problem->task[i].assigned_to=melhorAssignedTo[i];
    }
    int nrTasks=0;
    fprintf(fp, "----- Solução a contabilizar os lucros! -----\n");
    for(int p=0;p<problem->P;p++) {
        fprintf(fp,"\nPROGRAMADOR %d\n",(p+1));
            for(int t=0;t<problem->T;t++){
                if(problem->task[t].assigned_to==p){
                  fprintf(fp,"Foi atribuída a task que começa em %d e acaba em %d com
lucro                                                                                de
%d\n",problem->task[t].starting_date,problem->task[t].ending_date,problem->task[t].profit);
                  }
             }
       }
       fprintf(fp,"\nO profit total é %d\n\n",problem->total_profit);
    }
```

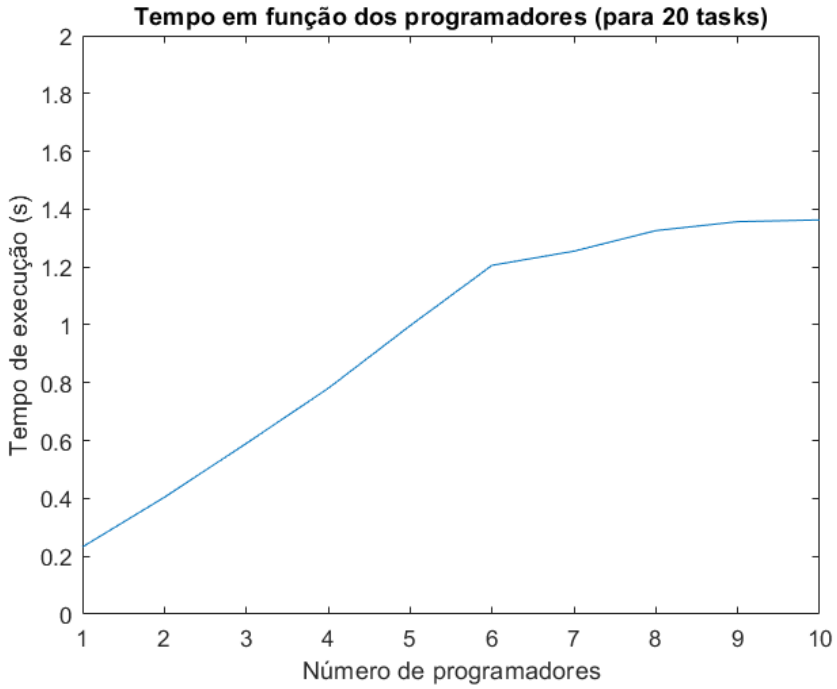


Fig.13 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de programadores (até 10 programadores) a realizar tarefas para um total de 20 tarefas.
[eixo dos y está numa escala de ordem 10^(1) segundos]

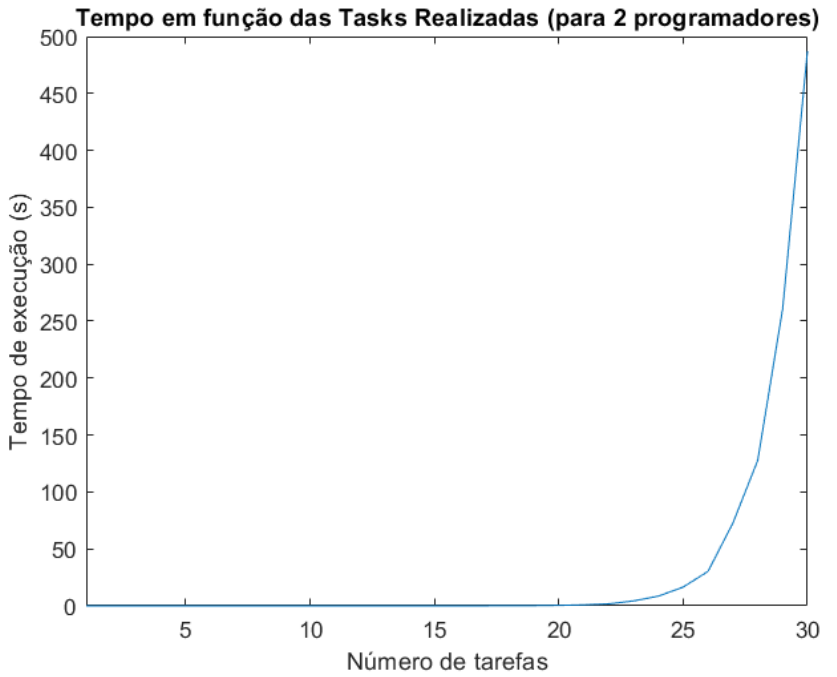
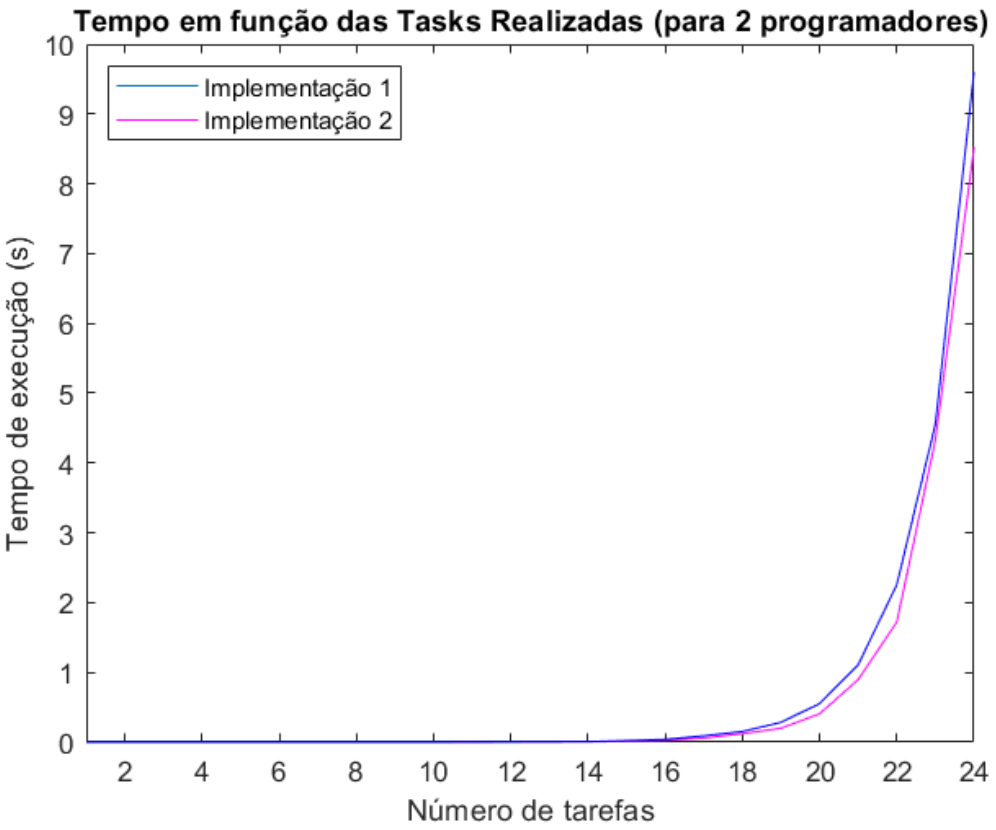


Fig.14 - Gráfico da variação do tempo de execução para esta implementação em relação ao aumento de tarefas (até 24 tarefas) a realizar por um total de 2 programadores.
[eixo dos y está numa escala de ordem 10^(1) segundos]

4.3 Comportamento de cada abordagem em relação ao tempo

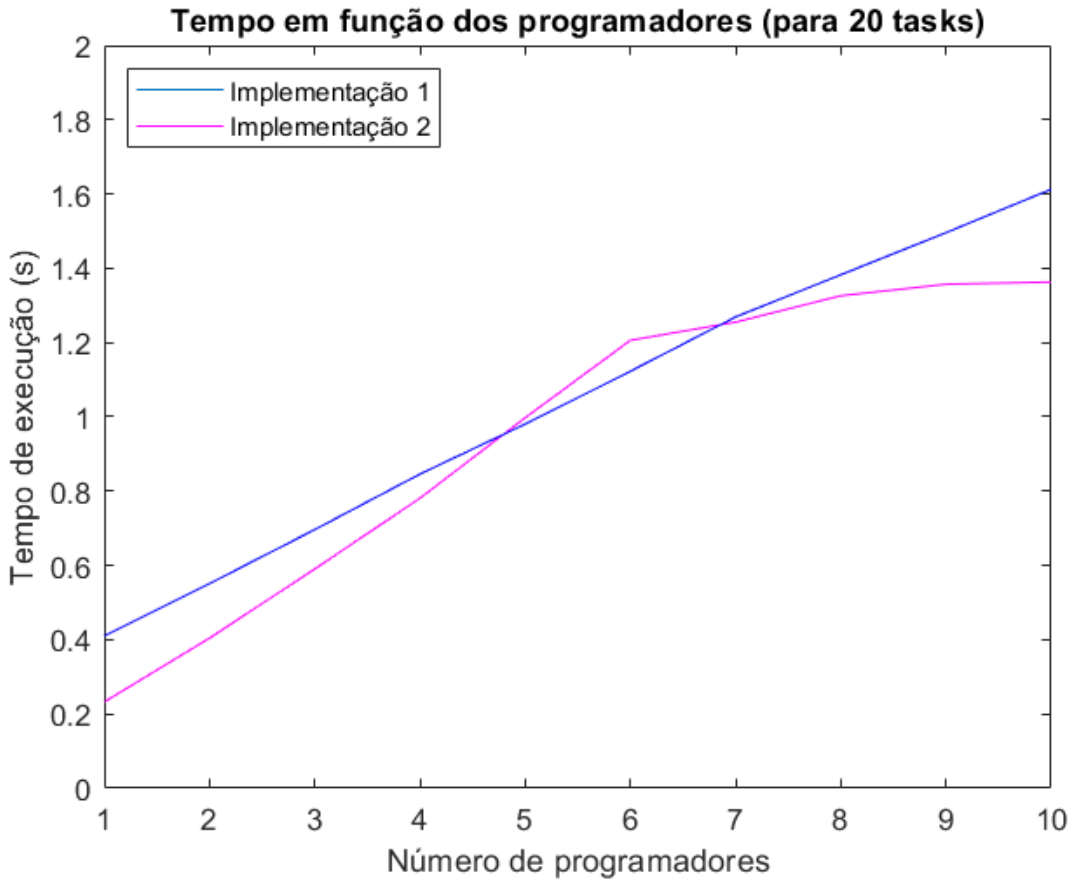
- **Para um número fixo de programadores**



Como é possível observar no gráfico de comparação dos tempos de execução entre a abordagem 1 (tópico 4.1) e a abordagem 2 (tópico 4.2) para um máximo de tarefas igual a 24 (uma vez que a abordagem 1 interrompe o processo quando o número de *tasks* é superior a 24) e um número de programadores constante igual a 2, a segunda implementação mostra-se mais eficiente que a primeira implementação. Isto ocorre porque as combinações geradas na segunda implementação não são guardadas em nenhuma estrutura e há uma validação da viabilidade da combinação.

-

- **Para um número fixo de tarefas**



Como é possível observar no gráfico de comparação dos tempos de execução entre a abordagem 1 (tópico 4.1) e a abordagem 2 (tópico 4.2) para um máximo de programadores igual a 10 e um número de tarefas constante igual a 20, a segunda implementação mostra-se mais eficiente que a primeira implementação. Isto ocorre porque as combinações geradas na segunda implementação não são guardadas em nenhuma estrutura e há uma validação da viabilidade da combinação.

- **Para os exemplos dos slides**

Observe o tempo de execução para a abordagem 2, que se revelou a melhor abordagem, para os exemplos demonstrados nos slides.

Nº de programadores	Nº de tarefas	Total de tarefas	Lucro total (valorizando o lucro)	Tempo de execução
4	20	9	35897	7.099e-01
6	30	13	48885	1.376e+03

5. Análise para os diversos números mecanográficos

Por último, neste tópico iremos analisar o comportamento do *script* com a implementação que se mostrou mais eficiente segundo as análises realizadas anteriormente, ou seja, a abordagem seis (sem lucro)/abordagem dois (com lucro) para os números mecanográficos dos elementos que constituem o grupo. Novamente, tanto os gráficos como as informações serão mostradas para um número máximo de 30 tarefas por questões de duração da execução do código.

-

- **Número mecanográfico: 98607**

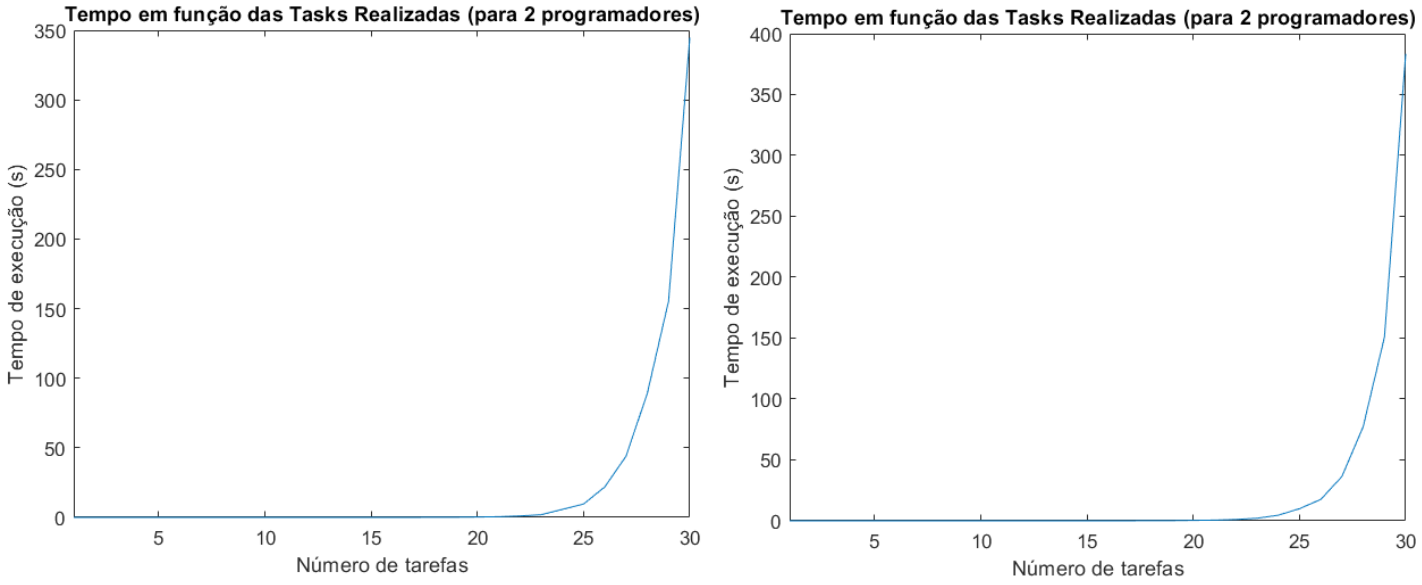


Fig.15 - Gráfico da variação do tempo de execução em relação ao aumento de tarefas a realizar por um total de 2 programadores. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda)
[eixo dos y está numa escala de ordem 10^(1) segundos]

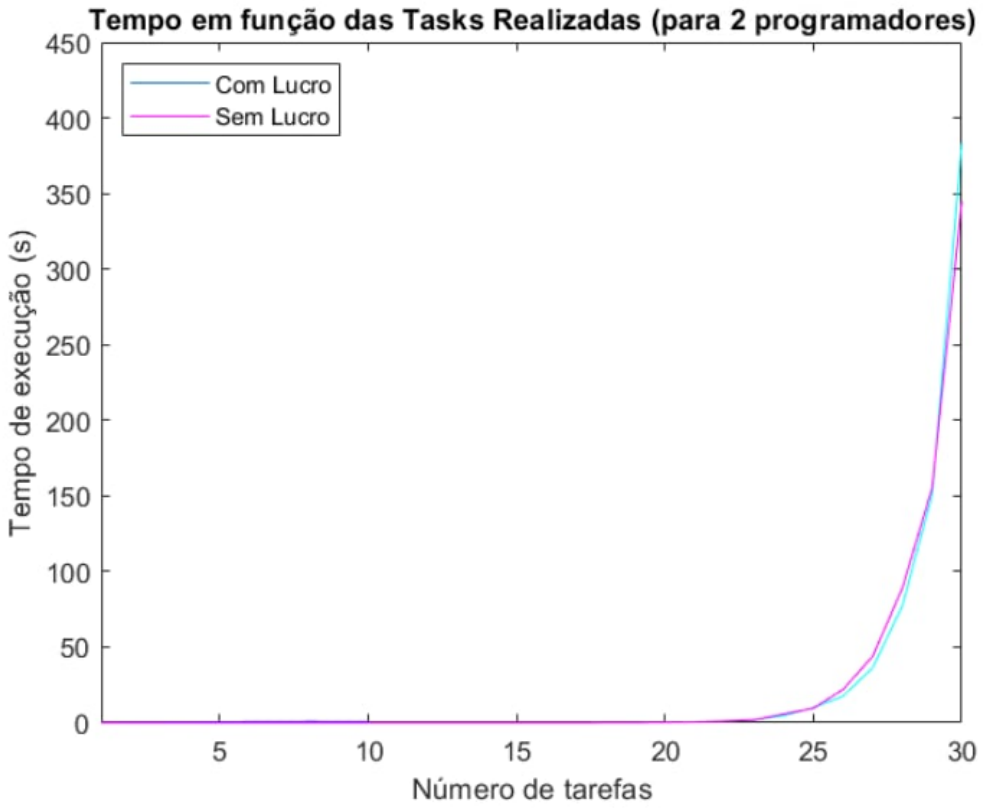


Fig.16 - Gráfico para comparação entre funções.

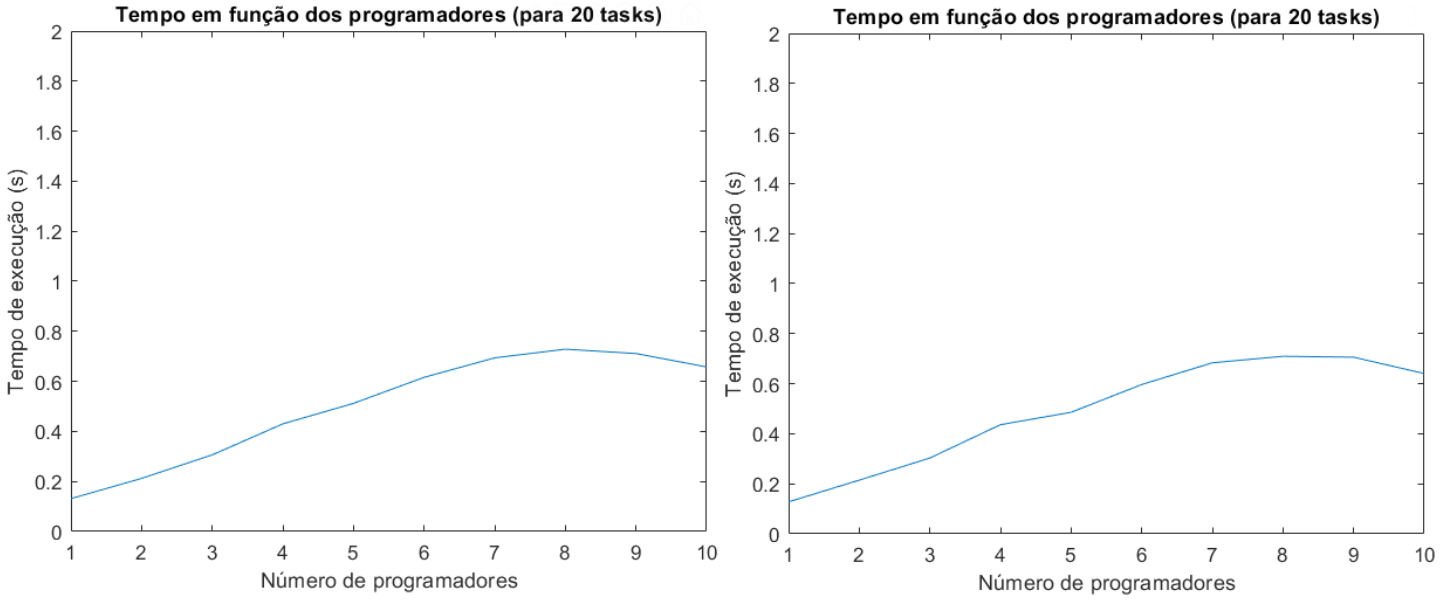


Fig.17 - Gráfico da variação do tempo de execução em relação ao aumento de programadores a realizar tarefas para um total de 20 tarefas. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda) [eixo dos y está numa escala de ordem 10^(1) segundos]

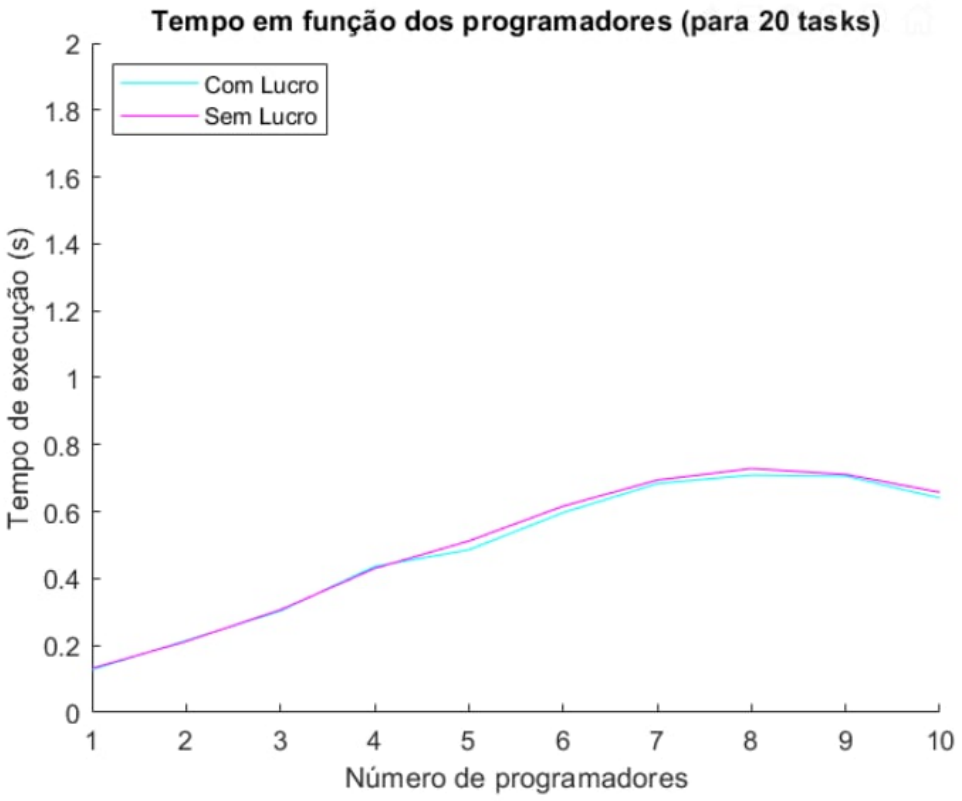


Fig.18 - Gráfico para comparação entre funções.

Nº de	Total de tarefas	Lucro total	Tempo de	Total de tarefas	Lucro total	Tempo de

tarefas	(não valorizando o lucro)	(não valorizando o lucro)	execução	(valorizando o lucro)	(valorizando o lucro)	execução
01	1	1	2.325e-05	1	3191	2.570e-05
02	2	2	2.382e-05	2	2528	2.433e-05
03	2	2	4.030e-05	2	5429	2.399e-05
04	2	2	4.575e-05	2	8679	2.440e-05
05	3	3	3.586e-05	2	8413	3.595e-05
06	2	2	5.616e-05	2	7245	4.538e-05
07	4	4	7.320e-05	4	6566	1.103e-04
08	4	4	1.190e-04	4	9367	1.227e-04
09	5	5	2.041e-04	4	10976	1.717e-04
10	6	6	3.715e-04	6	15388	4.035e-04
11	4	4	5.900e-04	4	15326	6.303e-04
12	6	6	1.118e-03	4	17341	1.618e-03
13	6	6	2.343e-03	3	19071	2.390e-03
14	8	8	4.311e-03	6	21665	4.875e-03
15	7	7	8.004e-03	5	19434	1.287e-02
16	7	7	1.852e-02	4	23586	2.128e-02
17	7	7	3.830e-02	5	35389	4.840e-02
18	11	11	7.582e-02	10	24063	8.285e-02
19	9	9	1.320e-01	6	27777	1.476e-01
20	10	10	2.809e-01	7	20564	3.434e-01
21	11	11	5.012e-01	7	26339	5.593e-01
22	10	10	1.049e+00	8	28267	1.064e+00
23	10	10	1.911e+00	6	26180	2.106e+00
24	13	13	5.736e+00	6	30996	4.599e+00
25	11	11	9.599e+00	5	26987	9.968e+00
26	10	10	2.196e+01	4	38844	1.756e+01

27	12	12	4.402e+01	8	32830	3.645e+01
28	17	17	8.918e+01	11	39133	7.743e+01
29	17	17	1.553e+02	12	39556	1.509e+02
30	16	16	3.447e2	7	39171	3.834e2

Tabela 01 - Tabela com as informações obtidas na execução do *script* para o número mecanográfico 98607, de forma a elaborar o gráfico tempo de execução em função do número de tarefas (número de programadores fixo com valor igual a 2).

Nº de programadores	Total de tarefas (sem valorizando o lucro)	Lucro total (não valorizando o lucro)	Tempo de execução	Total de tarefas (valorizando o lucro)	Lucro total (valorizando o lucro)	Tempo de execução
01	11	11	1.315e-01	10	21492	1.279e-01
02	10	10	2.809e-01	7	20564	3.434e-01
03	8	8	3.062e-01	7	22417	3.053e-01
04	13	13	4.302e-01	13	23116	4.363e-01
05	11	11	5.122e-01	6	22640	4.858e-01
06	10	10	6.162e-01	8	25785	5.968e-01
07	12	12	6.941e-01	10	27622	6.836e-01
08	12	12	7.287e-01	9	28848	7.096e-01
09	13	13	7.112e-01	13	27838	7.060e-01
10	14	14	6.578e-01	13	33362	6.407e-01

Tabela 02 - Tabela com as informações obtidas na execução do *script* para o número mecanográfico 98607, de forma a elaborar o gráfico tempo de execução em função do número de programadores (número de tarefas fixo com valor igual a 20).

- **Número mecanográfico: 98430**

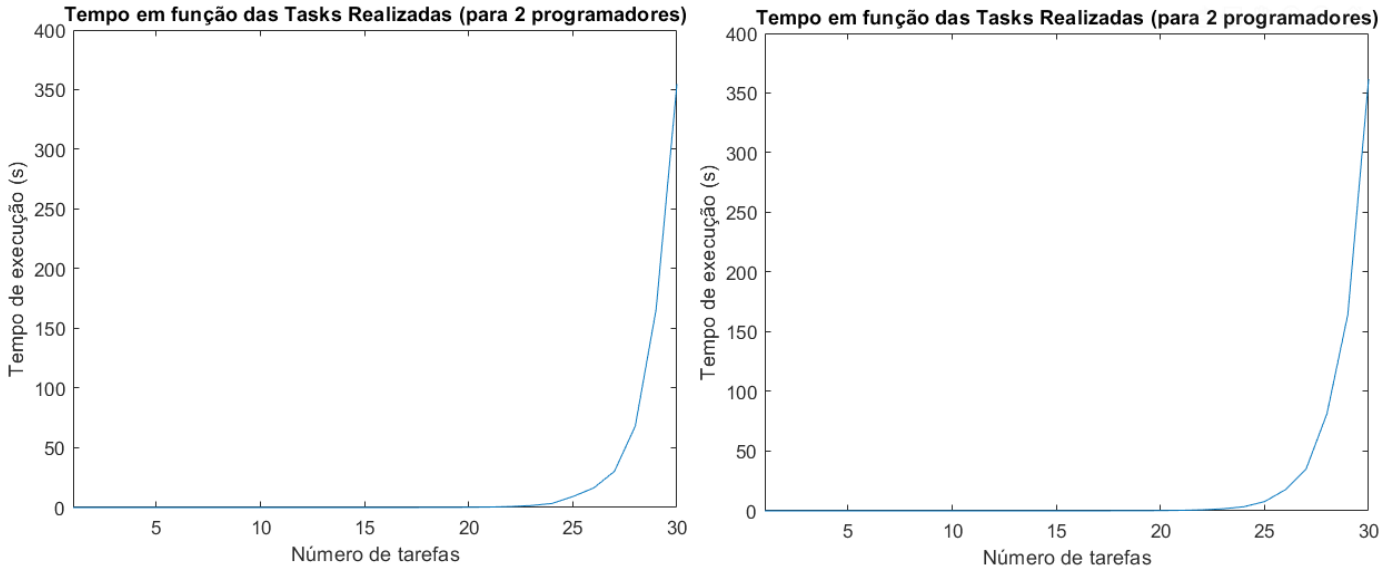


Fig.19 - Gráfico da variação do tempo de execução em relação ao aumento de tarefas a realizar por um total de 2 programadores. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda)
[eixo dos y está numa escala de ordem 10^(1) segundos]

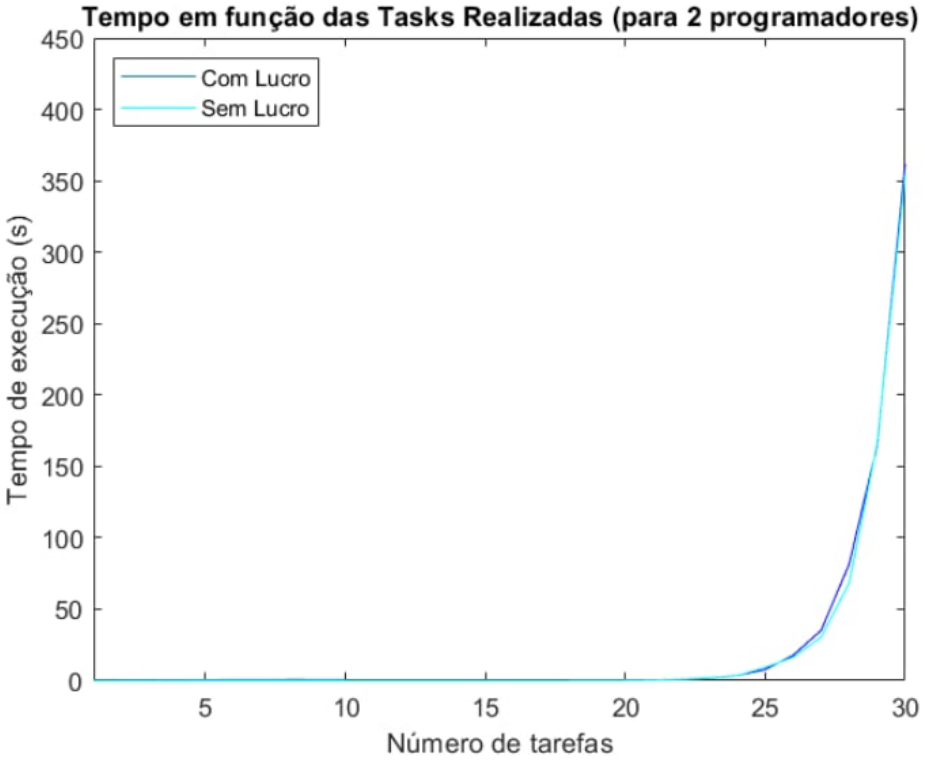


Fig.20 - Gráfico para comparação entre funções.

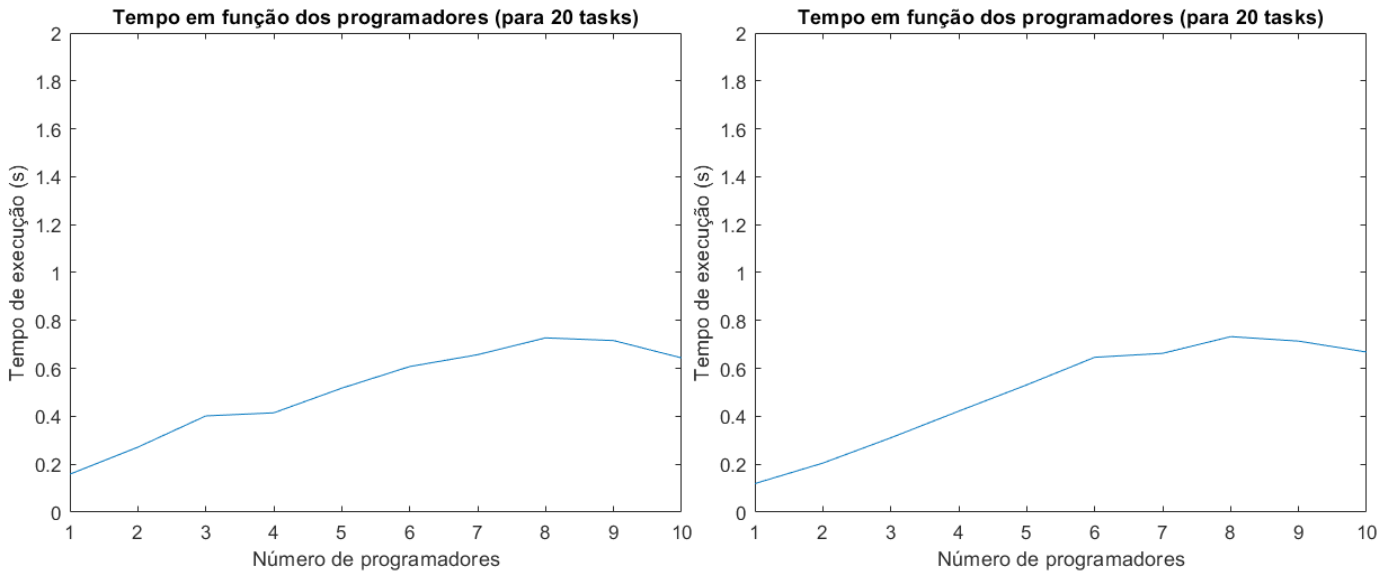


Fig.21 - Gráfico da variação do tempo de execução em relação ao aumento de programadores a realizar tarefas para um total de 20 tarefas. (contabilizando o lucro à esquerda e sem contabilizar o lucro à direita) [eixo dos y está numa escala de ordem 10^(1) segundos]

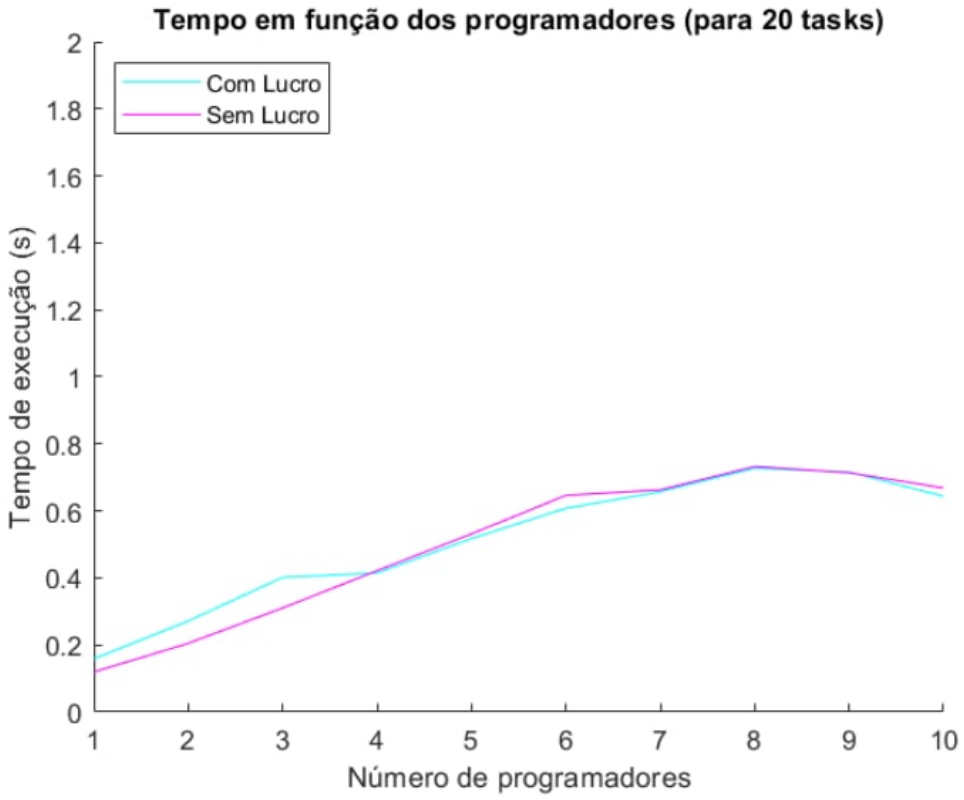


Fig.22 - Gráfico para comparação entre funções.

Nº de	Total de tarefas	Lucro total	Tempo de	Total de tarefas	Lucro total	Tempo de

tarefas	(não valorizando o lucro)	(não valorizando o lucro)	execução	(valorizando o lucro)	(valorizando o lucro)	execução
01	1	1	6.739e-05	1	1022	5.716e-05
02	2	2	3.431e-05	2	7766	7.918e-05
03	3	3	4.868e-05	3	4844	8.331e-05
04	2	2	2.458e-05	2	5123	8.893e-05
05	2	2	5.610e-05	2	8132	1.041e-04
06	2	2	8.066e-05	2	6902	1.307e-04
07	3	3	9.157e-05	3	7919	1.703e-04
08	6	6	1.655e-04	5	7745	3.237e-04
09	5	5	2.380e-04	2	10858	4.457e-04
10	5	5	3.417e-04	5	17121	9.223e-04
11	7	7	1.838e-03	6	14085	2.254e-03
12	7	7	3.539e-03	6	18362	3.564e-03
13	5	5	5.492e-03	5	15633	5.559e-03
14	7	7	7.441e-03	3	24250	7.733e-03
15	8	8	2.369e-02	5	17732	1.232e-02
16	9	9	2.208e-02	8	22221	2.167e-02
17	10	10	4.684e-02	2	34614	4.753e-02
18	9	9	6.684e-02	9	23253	6.624e-02
19	8	8	1.106e-01	6	22471	1.101e-01
20	8	8	2.038e-01	5	21235	2.057e-01
21	12	12	3.931e-01	3	35494	4.387e-01
22	11	11	8.273e-01	10	26578	8.509e-01
23	12	12	1.640e+00	6	32530	1.723e+00
24	15	15	3.390e+00	6	36374	3.460e+00
25	13	13	9.171e+00	14	25110	7.747e+00
26	13	13	1.636e+01	10	40784	1.770e+01
27	15	15	3.034e+01	12	32477	3.521e+01

28	17	17	6.822e+01	10	35939	8.150e+01
29	16	16	1.659e+02	9	35026	1.644e+02
30	17	17	3.547e+02	16	30962	3.618e+02

Tabela 03 - Tabela com as informações obtidas na execução do *script* para o número mecanográfico 98430, de forma a elaborar o gráfico tempo de execução em função do número de tarefas (número de programadores fixo com valor igual a 2).

Nº de programadores	Total de tarefas (sem valorizando o lucro)	Lucro total (não valorizando o lucro)	Tempo de execução	Total de tarefas (valorizando o lucro)	Lucro total (valorizando o lucro)	Tempo de execução
01	8	8	1.193e-01	4	30240	1.586e-01
02	8	8	2.038e-01	5	21235	2.057e-01
03	11	11	3.104e-01	10	21890	4.018e-01
04	11	11	5.307e-01	9	26202	5.255e-01
05	12	12	5.313e-01	9	25801	5.176e-01
06	13	13	6.465e-01	9	25233	6.078e-01
07	11	11	6.629e-01	10	27099	6.573e-01
08	12	12	7.328e-01	12	26970	7.275e-01
09	16	16	7.139e-01	15	26278	7.163e-01
10	15	15	6.685e-01	14	34140	6.438e-01

Tabela 04 - Tabela com as informações obtidas na execução do *script* para o número mecanográfico 98430, de forma a elaborar o gráfico tempo de execução em função do número de programadores (número de tarefas fixo com valor igual a 20).

-

- **Número mecanográfico: 98599**

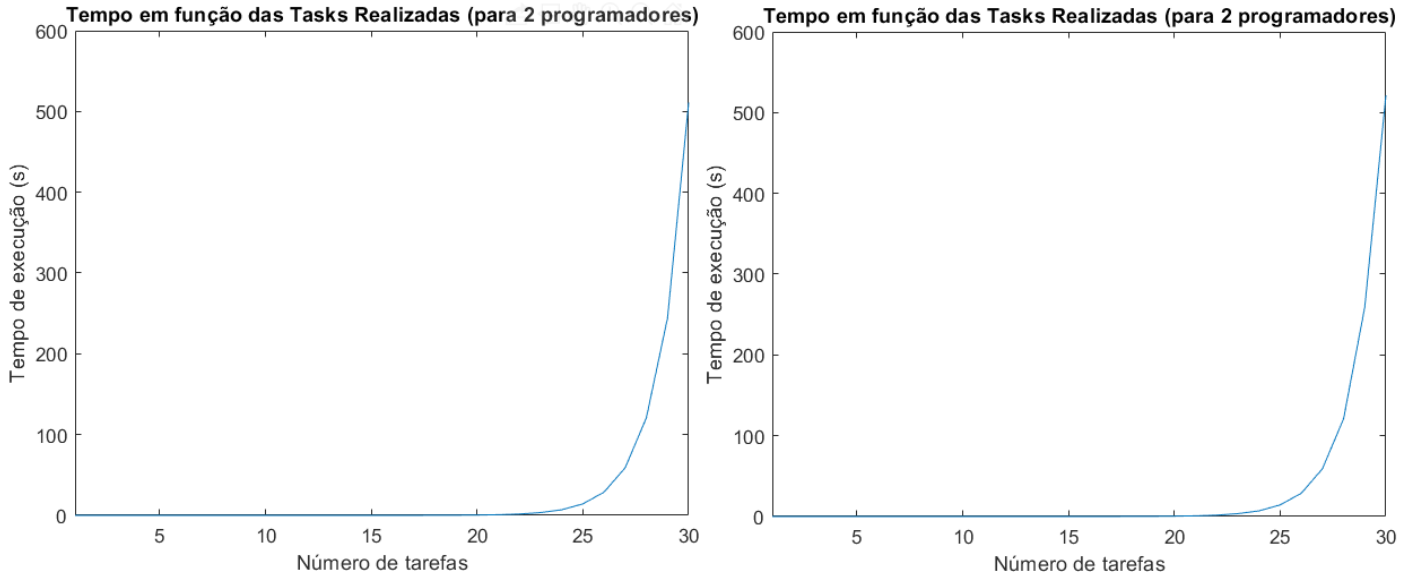


Fig.23 - Gráfico da variação do tempo de execução em relação ao aumento de tarefas a realizar por um total de 2 programadores. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda)
[eixo dos y está numa escala de ordem 10^(1) segundos]

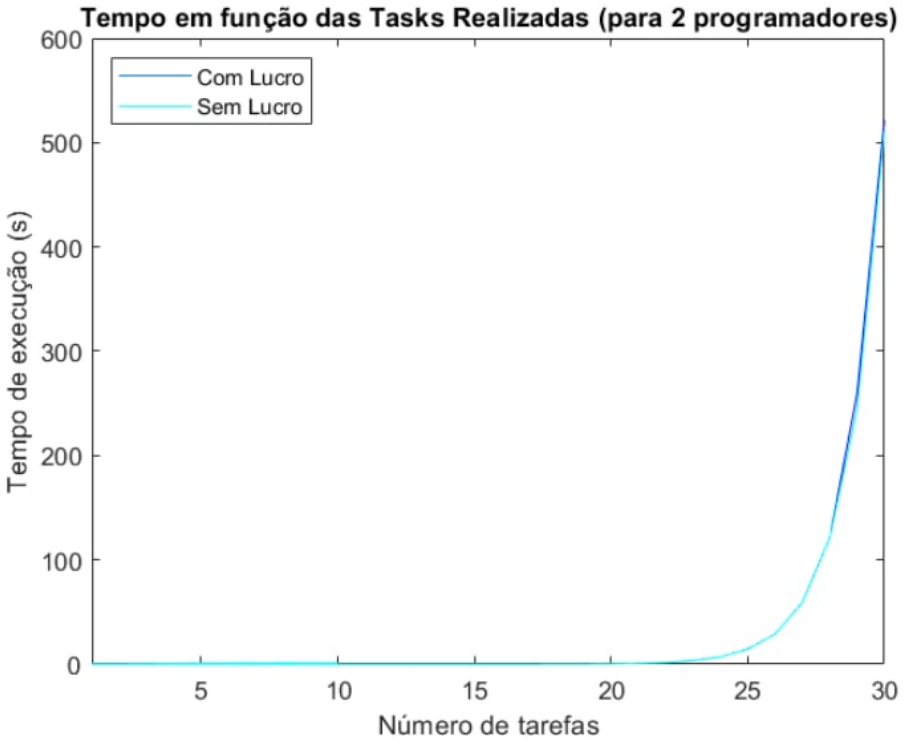


Fig.24 - Gráfico para comparação entre funções.

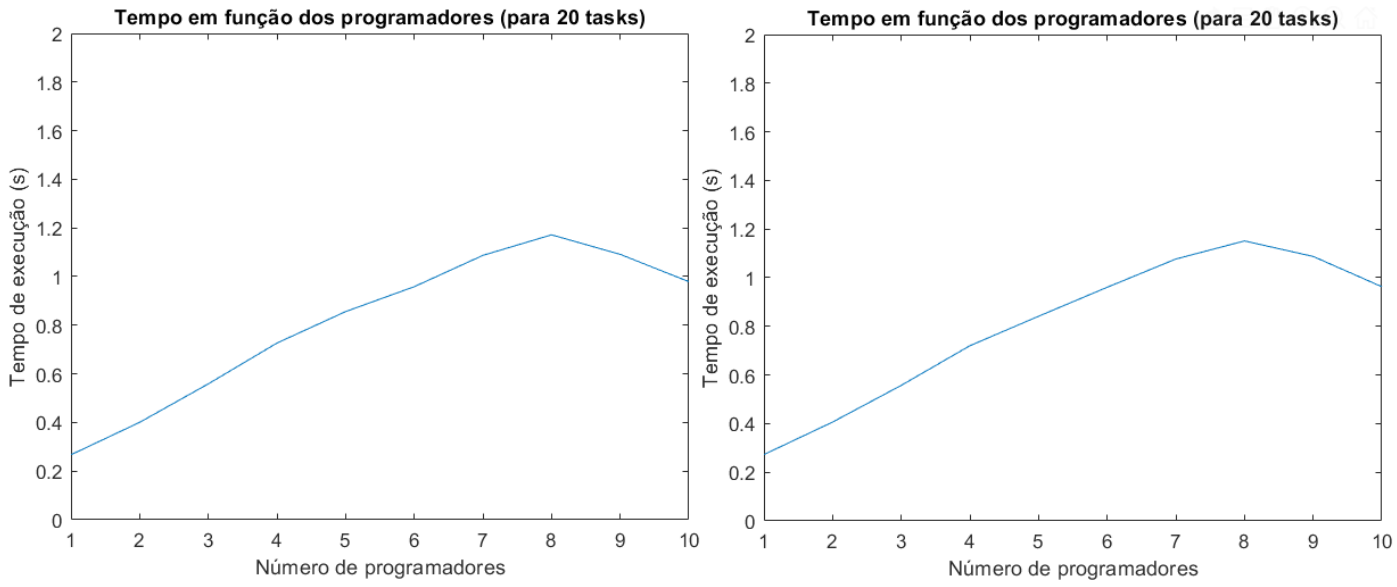


Fig.25 - Gráfico da variação do tempo de execução em relação ao aumento de programadores a realizar tarefas para um total de 20 tarefas. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda) [eixo dos y está numa escala de ordem 10^(1) segundos]

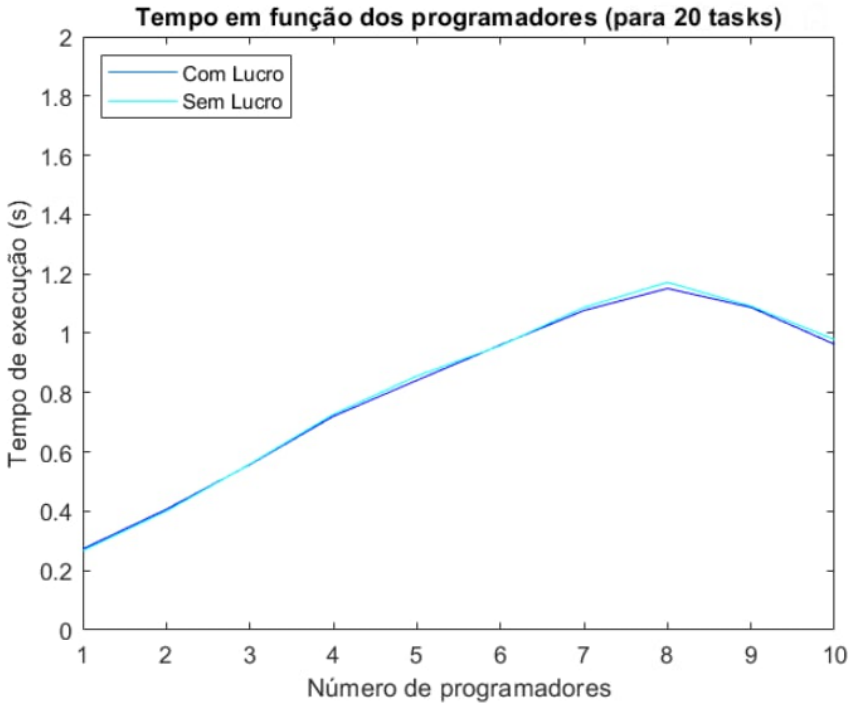


Fig.26 - Gráfico para comparação entre funções.

Nº de tarefas	Total de tarefas (sem valorizando o lucro)	Lucro total (não valorizando o lucro)	Tempo de execução	Total de tarefas (valorizando o lucro)	Lucro total (valorizando o lucro)	Tempo de execução
01	1	1	4.689e-05	1	2645	3.918e-05
02	2	2	6.472e-05	2	2546	4.371e-05
03	2	2	6.736e-05	2	6998	6.095e-05
04	2	2	4.880e-05	2	5182	7.468e-05
05	4	4	7.941e-05	4	5791	6.291e-05
06	4	4	1.118e-04	3	8774	8.471e-05
07	4	4	1.344e-04	4	8704	1.369e-04
08	3	3	1.694e-04	3	7211	2.265e-04
09	4	4	5.266e-04	3	10599	3.490e-04
10	5	5	9.264e-04	4	10743	9.645e-04
11	7	7	2.116e-03	6	10918	1.985e-03
12	5	5	3.023e-03	3	20063	3.638e-03
13	6	6	5.671e-03	5	14393	5.768e-03
14	7	7	1.087e-02	2	17538	1.136e-02
15	7	7	1.889e-02	5	18491	2.020e-02
16	8	8	3.332e-02	5	21600	3.427e-02
17	10	10	6.299e-02	9	14163	6.505e-02
18	10	10	6.299e-02	7	22270	1.124e-01
19	9	9	1.130e-01	8	24446	2.104e-01
20	9	9	3.975e-01	7	22807	4.089e-01
21	12	12	8.329e-01	8	30339	8.371e-01
22	9	9	1.648e+00	7	36858	1.659e+00
23	14	14	3.466e+00	7	31518	3.524e+00
24	14	14	6.948e+00	11	32640	6.953e+00
25	14	14	1.423e+01	6	36925	1.435e+01

26	15	15	2.881e+01	10	36021	2.880e+01
27	14	14	5.896e+01	8	43071	5.912e+01
28	12	12	1.207e+02	8	32918	1.210e+02
29	15	15	2.442e+02	7	45859	2.590e+02
30	17	17	5.110e+02	11	35164	5.213e+02

Tabela 05 - Tabela com as informações obtidas na execução do *script* para o número mecanográfico 98599, de forma a elaborar o gráfico tempo de execução em função do número de tarefas (número de programadores fixo com valor igual a 2).

Nº de programadores	Total de tarefas (sem valorizando o lucro)	Lucro total (não valorizando o lucro)	Tempo de execução	Total de tarefas (valorizando o lucro)	Lucro total (valorizando o lucro)	Tempo de execução
01	9	9	2.674e-01	1	32441	2.729e-01
02	9	9	4.007e-01	7	22807	4.065e-01
03	11	11	5.589e-01	9	21368	5.574e-01
04	13	13	7.266e-01	10	26573	7.204e-01
05	9	9	8.561e-01	8	28941	8.416e-01
06	8	8	9.585e-01	7	26144	9.608e-01
07	11	11	1.087e+00	11	23083	1.077e+00
08	17	17	1.172e+00	15	28428	1.151e+00
09	11	11	1.091e+00	10	31173	1.088e+00
10	14	14	9.793e-01	14	27138	9.632e-01

Tabela 06 - Tabela com as informações obtidas na execução do *script* para o número mecanográfico 98599, de forma a elaborar o gráfico tempo de execução em função do número de programadores (número de tarefas fixo com valor igual a 20).

- Comparação entre os diferentes números mecanográficos

Nº mec	Nº de prog.	Nº de tar.	Total de tar. (s/ lucro)	Lucro total (s/lucro)	Tempo de execução (s/ lucro)	Total de tar. (c/ lucro)	Lucro total (c/ lucro)	Tempo de execução (c/ lucro)
98607	4	20	13	13	4.302e-01	13	23116	4.363e-01
98607	6	30	18	18	8.863e+02	14	41452	8.491e+02
98430	4	20	11	11	5.307e-01	11	26202	5.255e-01
98430	6	30	18	18	1.040e+03	14	34890	8.652e+02
98599	4	20	13	13	7.334-01	10	26573	7.388e-01
98599	6	30	17	17	1.375e+03	14	38835	1.367e+03

Tabela 07 - Tabela com as informações obtidas na execução do *script* para os números mecanográfico 98607, 98430 e 98599 com os seguintes inputs “20 4 0”, “20 4 1”, “30 6 0”, “30 6 1”

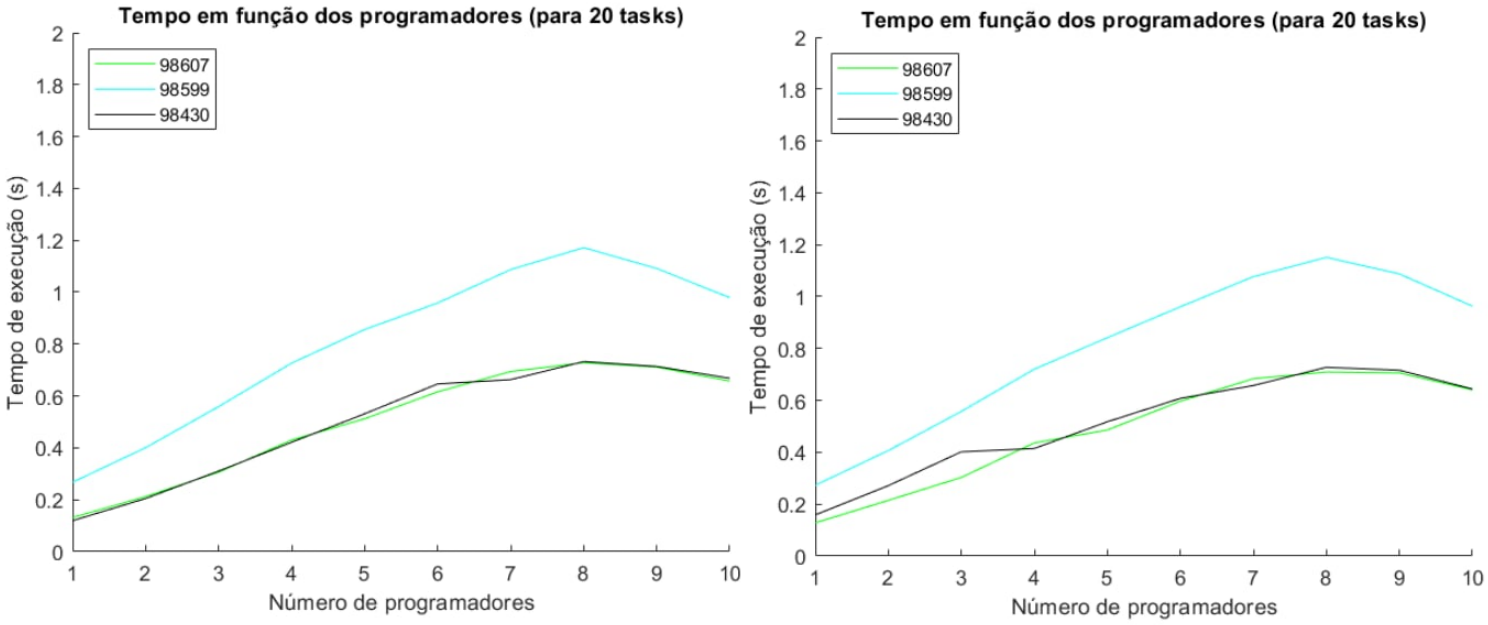


Fig.27 - Gráfico da variação do tempo de execução em relação ao aumento de programadores a realizar tarefas para um total de 20 tarefas. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda) [eixo dos y está numa escala de ordem 10^(1) segundos]

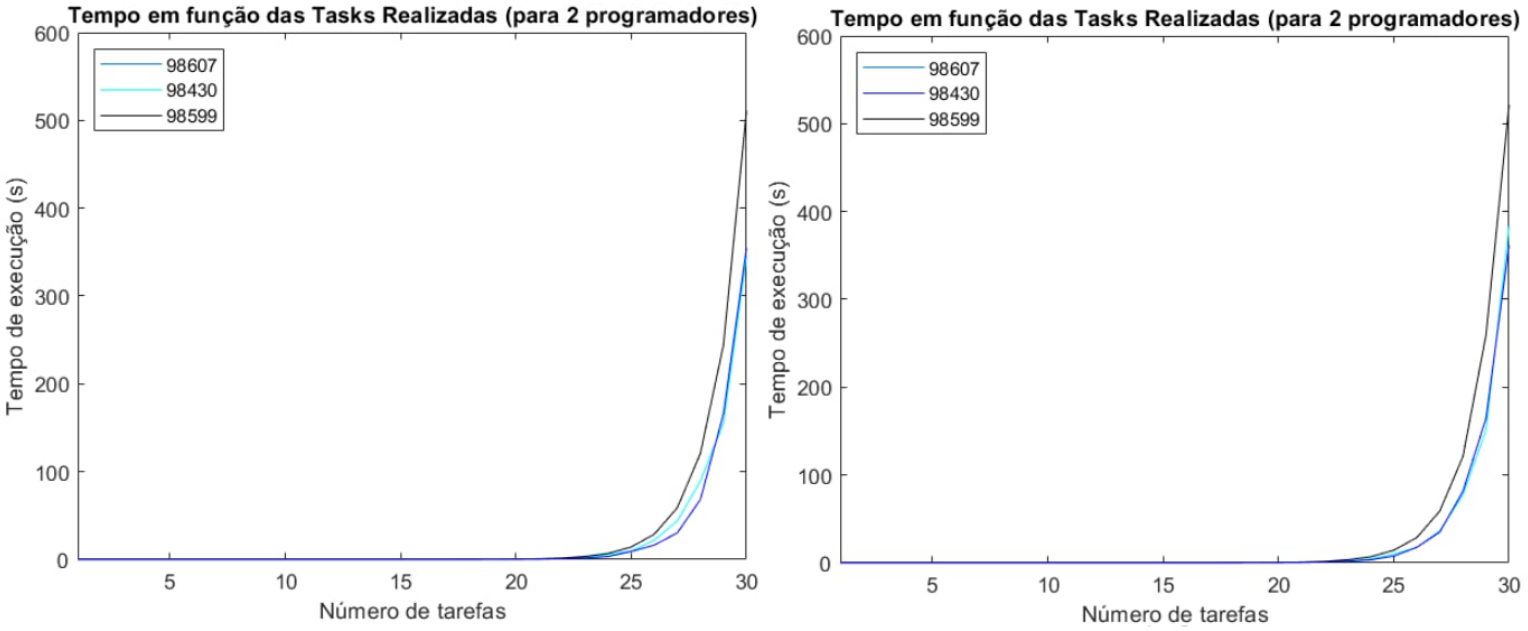


Fig.28 - Gráfico da variação do tempo de execução em relação ao aumento de tarefas a realizar por um total de 2 programadores. (contabilizando o lucro à direita e sem contabilizar o lucro à esquerda)
[eixo dos y está numa escala de ordem 10^(1) segundos]

6. Anexo

6.1 Código em C

```
////////////////////////////////////////
////////////////////////////////////////
//////////////////////////////
//
// AED, 2020/2021
//

// TODO: Diana Elisabete Siso Oliveira, nº
98607

// TODO: Miguel Rocha Ferreira, nº 98599

// TODO: Paulo Guilherme Soares Pereira, nº
98430

//

// 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 <sys/stat.h>

#include <sys/types.h>

#include "elapsed_time.h"

#include <unistd.h>

////////////////////////////////////////////
////////////////////////////////////////////
//////////////////////////////

//

// 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/p
rograms/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;

int compare_tasks_ending_2(const void
*t1,const void *t2){

        int d1,d2;

        d1 = ((task_t *)t1)->ending_date;

        d2 = ((task_t *)t2)->ending_date;

        if(d1 != d2)

                return (d1 < d2) ? -1 : +1;

        d1 = ((task_t *)t1)->starting_date;

        d2 = ((task_t *)t2)->starting_date;

        if(d1 != d2)

                return (d1 < d2) ? -1 : +1;

        return 0;

}
```

```
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;

}

int compare_tasks_ending(const void
*t1,const void *t2) {

        int d1,d2;

        d1 = ((task_t *)t1)->ending_date;

        d2 = ((task_t *)t2)->ending_date;

        if(d1 != d2)

                return (d1 > d2) ? -1 : +1;

        d1 = ((task_t *)t1)->starting_date;

        d2 = ((task_t *)t2)->starting_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
optionChosen) {

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) {

        printf(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

//OPÇÕES DE ORDENAÇÃO CONFORME OS INPUTS

if (problem->I == 1 && optionChosen == 3) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(problem->task[0]),compare_tasks_ending_2);

}

else if (problem->I == 0 && optionChosen ==
1) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(problem->task[0]),compare_tasks_ending);

}

else if (problem->I == 0 && optionChosen ==
2) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(problem->task[0]),compare_tasks_ending_2);

}

else if (problem->I == 1 && optionChosen ==
2) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(problem->task[0]),compare_tasks);

}

else if (problem->I == 1 && optionChosen ==
1) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(problem->task[0]),compare_tasks);

}

else if (problem->I == 1 && optionChosen ==
4) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(problem->task[0]),compare_tasks);

}

else if (problem->I == 1 && optionChosen ==
5) {

qsort((void
*)&problem->task[0],(size_t)problem->T,sizeo
f(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),"%06
d/%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

////////////////////////////////////////////
////////////////////////////////////////////
//////////////////////////////////

//

// problem solution (place your solution
here)

//FUNÇÕES!!

//FUNÇÃO 1 - INÍCIO

static int MelhorComb(int tarefa, int
**MatrizCompativeis, problem_t *problem, int
*jaEscolhida, int *tarefasProgramador, int
totalTasks) {

int counter = 0;

for (int element=tarefa; element<problem->T;
element++) {

if(MatrizCompativeis[tarefa][element] == 1)
{

int flag=0;

for (int e2=0; e2 <(counter); e2++) {

if
((MatrizCompativeis[jaEscolhida[e2]][element
] != 1) || (jaEscolhida[e2] == element)) {

flag = 1; break;

}

}

if (flag == 1) { continue; }

if (flag == 0){

jaEscolhida[counter] = element;

counter = counter + 1;

}

}

else { continue; }

}

for (int element=tarefa; element >= 0;
element--) {

if(MatrizCompativeis[tarefa][element] == 1)
{

int flag=0;

for (int e2=0; e2 <(counter); e2++) {

if
((MatrizCompativeis[jaEscolhida[e2]][element
] != 1) || (jaEscolhida[e2] == element)){

flag = 1; break;

}

}

if (flag == 1) { continue; }

if (flag == 0){

jaEscolhida[counter] = element;

counter = counter +1;

}
```

```
}

else { continue; }

}

if (totalTasks < counter) {

for (int x=0; x<problem->T; x++) {

tarefasProgramador[x] = -1;

}

totalTasks = counter;

for (int x=0; x<problem->T; x++) {

tarefasProgramador[x] = jaEscolhida[x];

}

}

return counter;

}

//FUNÇÃO 1 - FIM

//FUNÇÃO 2 - INÍCIO

static int funcaoTask(problem_t *problem,
int tarefa, int programador, int
nrTasksTotal) {

problem->busy[programador]=-1;

int *tarefasProgramador= (int *)
malloc(sizeof(int)*problem->T); //alocar
memória para o array tarefasprogramador

for(int m=0; m<problem->T;m++) //Inicializar
o vetor tarefasProgramador a -1 para cada
programador

{ tarefasProgramador[m]=-1; }

int nrTasks=0;

for(int k=tarefa; k<problem->T;k++){ //Para
cada task

if(problem->task[k].assigned_to==-1) {

if(problem->busy[programador]==-1) {//Caso o
programador esteja disponível

if(problem->task[k].assigned_to==-1){ //Caso
a task não esteja atribuída

tarefasProgramador[nrTasks]=k; //Guarda a
task num vetor

nrTasks++; //Incrementa o nr de tasks

problem->busy[programador]=problem->task[k].
ending_date; //Põe o programador busy até à
data de fim da tarefa

}

} else {

if(problem->task[k].starting_date>problem->b
usy[programador]) {//Caso a data de inicio
da tarefa seja maior que o fim da tarefa que
o programador está a fazer

problem->busy[programador]=problem->task[k].
ending_date; //Põe o programador busy até à
data de fim da tarefa

tarefasProgramador[nrTasks]=k; //Guarda a
task num vetor

nrTasks++;

}

}

}

problem->busy[programador] =
problem->task[tarefa].starting_date;

for (int element = tarefa; element>0;
element--) {

if(problem->task[element].assigned_to==-1) {

if
(problem->task[element].ending_date<problem-
>busy[programador]) {

tarefasProgramador[nrTasks]=element;

problem->busy[programador]=problem->task[ele
ment].starting_date;

nrTasks++;

}

}

}

if(nrTasks>nrTasksTotal) {//Caso o nr de
tasks seja maior que o total

for(int i=0;i<problem->T;i++) {//Para cada
task
```

```
if(problem->task[i].assigned_to==programador
) {//Caso alguma task esteja atribuida ao
programador p é eliminado o registo

problem->task[i].assigned_to=-1;

}

}

nrTasksTotal=nrTasks; //Nr tasks total passa
a ser o nr tasks

for(int i=0; i<problem->T;i++){ //Para cada
task

if(tarefasProgramador[i]!=-1) {//Se o vetor
tarefas tiver tarefas

problem->task[tarefasProgramador[i]].assigne
d_to=programador; //Atribui o valor ao
assigned to

}

}

}

return nrTasks; //NR Tasks maximo ate ao
momento que o programador pode fazer

}

//FUNÇÃO 2 - FIM

static int nrComb=0;

//FUNÇÃO 3 - INÍCIO

void addToArray(int arr[], int n,
int**combinacoes) //Função que coloca as
combinações no array combinações. Cada vez
que é usada a variável nrComb
{ //é incrementada e funciona como índice
para cada combinação. O seu valor final é o
nr de combinações.

for (int i = 0; i < n; i++) { //O valor
atribuído é 1 se a tarefa for para ser feita
e 0 se não for para ser feita

combinacoes[nrComb][i]=arr[i];

// printf("%d ",arr[i]);

}

nrComb++;

//printf("\n");

}

//FUNÇÃO 3 - FIM

//FUNÇÃO 4 - INÍCIO

void gerarCombinacoes(int n, int arr[], int
i,int **combinacoes) //Função que gera as
combinações binárias

{

if (i == n) {

addToArray(arr, n,combinacoes);

return;

}

arr[i] = 0;

gerarCombinacoes(n, arr, i + 1,combinacoes);

arr[i] = 1;

gerarCombinacoes(n, arr, i + 1,combinacoes);

}

//FUNÇÃO 4 - FIM

//FUNÇÃO 5 - INÍCIO

void function(int *comb, int
**tarefasProgramador, problem_t *problem,
int *melhorAssignedTo) {

static int nrTasksGeral=0; //Define-se a
variável nrTasksGeral (que vai guardar o
melhor número de tasks realizadas possível)
a 0

int profitAtual = 0; //Inicializa-se a
variável profitAtual(que vai guardar o
profit)

static int profitGeral=0;

int nrTasks=0;

int stop = 0;

for(int i=0;i<problem->P;i++){ //Inicializar
o vetor tarefasProgramador a -1

problem->busy[i]=-1;

for(int k=0;k<problem->T;k++){

tarefasProgramador[i][k]=-1;

if (stop < problem->T) {

problem->task[k].assigned_to=-1;

stop++;

}
```

```
}

}

for(int prog=0;prog<problem->P;prog++)
{//Para cada programador

for(int tar=0;tar<problem->T;tar++) {

if(comb[tar]==1) {//Caso a tarefa tar da
combinacao comb possa ser feita

if(problem->busy[prog]==-1) { //Caso o
programador esteja disponível

if(problem->task[tar].assigned_to==-1) {

tarefasProgramador[prog][nrTasks]=tar;

problem->busy[prog]=problem->task[tar].endin
g_date;

problem->task[tar].assigned_to=prog;

nrTasks++;

profitAtual=profitAtual+problem->task[tar].p
rofit;

}

}

else { //Caso o programador esteja ocupado

if(problem->task[tar].assigned_to==-1){

if(problem->busy[prog]<problem->task[tar].st
arting_date){

tarefasProgramador[prog][nrTasks]=tar;

problem->busy[prog]=problem->task[tar].endin
g_date;

problem->task[tar].assigned_to=prog;

nrTasks++;

profitAtual=profitAtual+problem->task[tar].p
rofit;

}

}

}

}

int flagE=0;

for(int i=0;i<problem->T;i++) {//Para cada
elemento da combinacao

if(comb[i]==1) {//Caso a task tenha que ser
realizada

if(problem->task[i].assigned_to==-1) {//Caso
a task não tenha sido atribuída

flagE=1;

break;

}

}

}

if(flagE==0){ //Caso todas as tasks que
tinham que ser feitas tiverem sido feitas

if(profitAtual>profitGeral) {

for(int i=0;i<problem->T;i++) {
//Inicializar o vetor melhorAssignedTo a -1

melhorAssignedTo[i]=-1;

}

nrTasksGeral=nrTasks;

profitGeral=profitAtual;

for(int i=0;i<problem->P;i++) {

for(int k=0;k<problem->T;k++) {

if(tarefasProgramador[i][k]!=-1) { //Caso o
elemento seja uma tarefa

melhorAssignedTo[tarefasProgramador[i][k]]=i
;

}

}

}

}

problem->total_profit=profitGeral;

}

//FUNÇÃO 5 - FIM
```

```
//FUNÇÃO 6 - INÍCIO

void generateAllBinaryStrings(int n, int
arr[], int i, int **tarefasProgramador, int
*melhorAssignedTo, problem_t *problem)
//Função que gera as combinações binárias

{

if (i == n) {

if (problem->T < problem->P) {

function(arr, tarefasProgramador, problem,
melhorAssignedTo);

}

else {

int contadorUm = 0;

for (int a=0; a<problem->T; a++) {

if (arr[a] == 1) {

contadorUm++;

}

}

if (contadorUm >= problem->P) {

function(arr, tarefasProgramador, problem,
melhorAssignedTo); //quando já fez a
combinação ele corre a função

}

}

return;

}

arr[i] = 0;

generateAllBinaryStrings(n, arr, i + 1,
tarefasProgramador, melhorAssignedTo,
problem);

arr[i] = 1;

generateAllBinaryStrings(n, arr, i + 1,
tarefasProgramador, melhorAssignedTo,
problem);

}

//FUNÇÃO 6 - FIM

//FUNÇÃO 7 - SOLVE

static void solve(problem_t *problem, int
optionChosen)

{

FILE *fp;

int i;

(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();

printf("Aguarde...\n");

if (problem->I == 1) {

if (optionChosen == 1) {

fprintf(fp, "----- Solução a ignorar os
lucros! -----\n");

int numeroTasksTotal;

int numeroTasksProgramador;

for (int i=0;i<problem->T;i++) {
//Inicializar os vetores assigned to com T
tarefas

problem->task[i].assigned_to=-1;

}

for (int i=0;i<problem->P;i++){
//Inicializar o vetor Busy com P espaços

problem->busy[i]=-1;

}

int quantasTasks = 0;

for(int p=0; p<problem->P; p++) { //Para
cada programador

fprintf(fp,"\nPROGRAMADOR %d\n",(p+1));

numeroTasksProgramador=0; //Numero de tasks
que cada programador vai fazer
```

```
numeroTasksTotal=0; //Melhor numero de tasks
que cada programador vai fazer

for(int t=0;t<problem->T;t++) {//Para cada
task de início (o programa vai correr a
comecar em 0, 1, 2...)

numeroTasksProgramador=funcaoTask(problem,t,
p,numeroTasksTotal); //Chama a função

if(numeroTasksProgramador>numeroTasksTotal){
//Se o numero de tasks devolvido pela funcao
for maior substitui o nr de tasks total

numeroTasksTotal=numeroTasksProgramador;

}

}

for(int i=0;i<problem->T;i++){

if(problem->task[i].assigned_to==p){

quantasTasks++;

fprintf(fp, "Tarefa que começa em %d e acaba
em %d\n", problem->task[i].starting_date,
problem->task[i].ending_date);

}

}

}

printf("Numero de tasks: %d\n",
quantasTasks);

}

if (optionChosen == 2) {
```

```
fprintf(fp, "----- Solução a ignorar os
lucros! -----\n");

int **MatrizCompativeis;

MatrizCompativeis = malloc(problem->T *
sizeof(int*));

for (int i=0; i<problem->T; i++) {
MatrizCompativeis[i] = malloc(problem->T *
sizeof(int)); }

for (int t1=0; t1<problem->T; t1++) {

for (int t2=t1; t2 < problem->T; t2++) {

if (t1==t2) { MatrizCompativeis[t1][t2] = 1;
MatrizCompativeis[t2][t1] = 1; }

else {

if (problem->task[t1].ending_date <
problem->task[t2].starting_date) {
MatrizCompativeis[t1][t2] = 1;
MatrizCompativeis[t2][t1] = 1; }

else { MatrizCompativeis[t1][t2] = 0;
MatrizCompativeis[t2][t1] = 0; }

}

}

}

int nrTasks = 0;

for(int p=0; p<problem->P; p++) {

int *tarefasProgramador=(int *)
malloc(sizeof(int)*(problem->T));

int *jaEscolhida=(int *)
malloc(sizeof(int)*(problem->T));
```

```
for(int m=0; m<problem->T;m++) {
tarefasProgramador[m]=-1; }

fprintf(fp, "\nPROGRAMADOR %d\n", p+1);

int totalTasks = 0;

int tasksValidas = 0;

for (int tarefa=0; tarefa<problem->T;
tarefa++) {

for (int i=0; i<problem->T; i++) {
jaEscolhida[i]=-1; }

tasksValidas = MelhorComb(tarefa,
MatrizCompativeis,problem,jaEscolhida,tarefa
sProgramador,totalTasks);

if (tasksValidas > totalTasks) {
totalTasks=tasksValidas;}

}

for (int x=0; x<problem->T; x++) {

if (tarefasProgramador[x] != -1) {

nrTasks++;

fprintf(fp, "Tarefa que começa em %d e acaba
em %d\n",
problem->task[tarefasProgramador[x]].startin
g_date,
problem->task[tarefasProgramador[x]].ending_
date);

for (int coluna = 0;
coluna<problem->T;coluna++) {
MatrizCompativeis[tarefasProgramador[x]][col
una] = 0;
MatrizCompativeis[coluna][tarefasProgramador
[x]] = 0; }
```

```
}

}

free(tarefasProgramador);

free(jaEscolhida);

}

printf("Numero de tasks: %d\n", nrTasks);

free(MatrizCompativeis);

}

if (optionChosen == 3) {

int *Comb=(int *)
malloc(sizeof(int)*(problem->T));

int counter = 0;

for (int i=0; i<problem->T; i++) {
problem->task[i].assigned_to=-1; }

for (int p=1; p<=problem->P; p++){

for (int i=0; i<problem->T; i++) { Comb[i] =
-1; }

for (int task=0; task<problem->T; task++) {

if (problem->task[task].assigned_to!= -1) {
continue; }

if (problem->busy[p] == -1) {

problem->task[task].assigned_to=p;

problem->busy[p]=problem->task[task].ending_
date;

Comb[counter]=task;

counter++;

}

else {

if
(problem->busy[p]<problem->task[task].starti
ng_date) {

problem->task[task].assigned_to=p;

problem->busy[p]=problem->task[task].ending_
date;

Comb[counter]=task;

counter++;

}

}

}

for (int task=0; task<counter; task++) {

int duration1 =
problem->task[Comb[task]].ending_date -
problem->task[Comb[task]].starting_date;

for (int element=0; element<problem->T;
element++) {

int duration2=
problem->task[element].ending_date -
problem->task[element].starting_date;

if (problem->task[element].assigned_to ==-1)
{

if (task == 0) {

if ((problem->task[element].ending_date <
problem->task[Comb[task+1]].starting_date)
&& (duration2>duration1)){

problem->task[Comb[task]].assigned_to=-1;

problem->task[element].assigned_to= p;

Comb[task] = element;

}

}

else if (task== (counter -1)) {

if ((problem->task[element].starting_date >
problem->task[Comb[task-1]].ending_date) &&
(duration2>duration1)){

problem->task[Comb[task]].assigned_to=-1;

problem->task[element].assigned_to= p;

Comb[task] = element;

}

}

else {

if ((problem->task[element].starting_date >
problem->task[Comb[task-1]].ending_date) &&
(problem->task[element].ending_date <
problem->task[Comb[task+1]].starting_date)
&& (duration2>duration1)){

problem->task[Comb[task]].assigned_to=-1;

problem->task[element].assigned_to= p;
```

```
Comb[task] = element;

}

}

}

}

}

fprintf(fp,"\nPROGRAMADOR: %d\n", p);

for (int task = 0; task<problem->T; task++)
{

if (Comb[task] != -1) {

fprintf(fp,"Tarefa que começa em %d e acaba
em %d\n",
problem->task[Comb[task]].starting_date,
problem->task[Comb[task]].ending_date);

}

}

}

free(Comb);

}

if (optionChosen == 4) {

int i;

int nrTasks;

int nrTasksGeral;
```

```
int *melhorAssignedTo; //Array que guarda a
melhor combinação de atribuição de tasks

int **combinacoes; //Array de arrays que
guarda todas as combinações binárias (cada
array é uma combinação)

int **tarefasProgramador; //Array de arrays
que guarda as tarefas que cada programador
faz numa dada combinação

combinacoes=(int**)malloc(sizeof(int*)*pow(2
,problem->T)); //Alocar espaço para o array
de arrays 'combinacoes'. O espaço é 2^T
porque existem 2^T combinações

for(int i=0;i<pow(2,problem->T);i++) //visto
que cada task pode ou não ser feita

{

combinacoes[i]=(int*)malloc(sizeof(int)*prob
lem->T);

}



tarefasProgramador=(int**)malloc(sizeof(int*
)*problem->P); //Alocar espaço para o array
de arrays 'tarefasProgramador'

for(int i=0;i<problem->P;i++)

{

tarefasProgramador[i]=(int*)malloc(sizeof(in
t)*problem->T);

}
```

```
melhorAssignedTo=(int*)malloc(sizeof(int)*pr
oblem->T); //Alocar espaço para o array
'melhorAssignedTo'



int n = problem->T; //Define a variável n
com o número de tarefas

int arr[n]; //Inicializa o array 'arr' com n
espaços



for(int i=0;i<pow(2,problem->T);i++)
//Atribui o valor -1 a todas as posições do
array de arrays 'combinacoes'

{

for(int k=0;k<problem->T;k++)

{

combinacoes[i][k]=-1;

}

}

gerarCombinacoes(n, arr, 0,combinacoes);
//Chama a função que gera as combinações
binárias



for(int i=0;i<problem->T;i++) //Inicializar
o vetor melhorAssignedTo a -1
```

```
{

melhorAssignedTo[i]=-1;

}



nrTasksGeral=0; //Define-se a variável
nrTasksGeral (que vai guardar o melhor
número de tasks realizadas possível) a 0

for(int
comb=0;comb<pow(2,problem->T);comb++) //Para
cada combinação

{



for(int i=0;i<problem->P;i++) //Inicializar
o vetor busy a -1

{

problem->busy[i]=-1;

}



for(int i=0;i<problem->P;i++) //Inicializar
o vetor tarefasProgramador a -1

{

for(int k=0;k<problem->T;k++)

{

tarefasProgramador[i][k]=-1;

}

}



for(int i=0;i<problem->T;i++) //Inicializar
o vetor assignedTo a -1

{

problem->task[i].assigned_to=-1;

}



nrTasks=0; //Variável nrTasks (que armazena
o nr de tasks realizadas para cada
combinação é inicializada a 0 para cada
combinação)



for(int prog=0;prog<problem->P;prog++)
//Para cada programador

{

for(int tar=0;tar<problem->T;tar++) //Para
cada tarefa da combinação comb

{

if(combinacoes[comb][tar]==1) //Caso a
tarefa tar da combinacao comb possa ser
feita

{

if(problem->busy[prog]==-1) //Caso o
programador esteja disponível

{

if(problem->task[tar].assigned_to==-1)
//Caso a tarefa não esteja atribuída

{

tarefasProgramador[prog][nrTasks]=tar;
//Guarda-se o valor de tar em
'tarefasProgramador'

problem->busy[prog]=problem->task[tar].endin
g_date; //Define-se o programador busy até
ao final dessa tarefa

problem->task[tar].assigned_to=prog;
//Define-se que a tarefa tar está atribuída
ao programador prog

nrTasks++; //Incrementa-se o número de tasks
realizadas na combinação atual

}

}



else //Caso o programador esteja ocupado

{

if(problem->task[tar].assigned_to==-1)
//Caso a tarefa não esteja atribuída

{

if(problem->busy[prog]<problem->task[tar].st
arting_date) //Se a starting date da tarefa
```

```
for maior que o valor de busy (ending date
da anterior)

{ //então o programador pode realizá-la
porque não vai haver sobreposição

tarefasProgramador[prog][nrTasks]=tar;
//Guarda-se o valor de tar em
'tarefasProgramador'

problem->busy[prog]=problem->task[tar].endin
g_date; //Define-se o programador busy até
ao final dessa tarefa

problem->task[tar].assigned_to=prog;
//Define-se que a tarefa tar está atribuída
ao programador prog

nrTasks++; //Incrementa-se o número de tasks
realizadas na combinação atual

}

}

}

}

}

}

int flagE=0; //A variável flagE vai servir
como detetora de inviabilidades. É definida
a zero aqui e vai passar por algumas
condições.

//Caso o seu valor se mantenha a zero quer
dizer que a combinação é viável. Caso o seu
valor se altere para 1 quer dizer que

//a combinação não reune as condições
necessárias para ser viável

//Para ser viável, todas as tasks que na
combinação tenham o valor '1' têm que ser
atribuídas. Caso haja tarefas que tenham o
valor '1'

//na combinação mas que não tenham sido
atribuídas, quer dizer que não há
programadores suficientes para as realizar a
todas, então

//a combinação revela-se inviável



for(int i=0;i<problem->T;i++) //Para cada
elemento da combinacao

{

if(combinacoes[comb][i]==1) //Caso a task
tenha que ser realizada

{

if(problem->task[i].assigned_to==-1) //Caso
a task não tenha sido atribuída

{

flagE=1; //A combinação é inviável

break;

}

}

}

if(flagE==0) //Caso todas as tasks que
tinham que ser feitas tiverem sido feitas

{

if(nrTasks>nrTasksGeral) //Se o número de
tasks desta combinação for superior ao nr de
tasks geral

{

for(int i=0;i<problem->T;i++) //Reicializar
o vetor melhorAssignedTo a -1

{

melhorAssignedTo[i]=-1;

}



nrTasksGeral=nrTasks; //Atualiza-se o valor
de nrTasksGeral para que contenha agora o
melhor número de tasks



for(int i=0;i<problem->P;i++) //Para cada
programador

{

for(int k=0;k<problem->T;k++) //Para cada
tarefa

{
```

```
if(tarefasProgramador[i][k]!=-1) //Caso o elemento seja uma tarefa

{

melhorAssignedTo[tarefasProgramador[i][k]]=i; //Atribui ao melhorAssignedTo, na posição correspondente à tarefa 'tarefasProgramador[i][k]'

} //o valor de i (do programador)

}

}

}

}

}

for(int i=0;i<problem->T;i++) //No final de todas as combinações, copiamos o valor do array melhorAssignedTo para o array assignedTo do problema

{

problem->task[i].assigned_to=melhorAssignedTo[i];

}



//Agora imprimem-se os resultados no ficheiro

fprintf(fp, "----- Solução a ignorar os lucros! -----\n");

for(int p=0;p<problem->P;p++) //Para cada programador

{

fprintf(fp,"\nPara o Programador %d\n",(p+1));

for(int t=0;t<problem->T;t++) //Para cada tarefa

{

if(problem->task[t].assigned_to==p) //Imprimem-se, para cada p, os valores das tarefas atribuídas a este

{

fprintf(fp,"Foi atribuída a task que começa em %d e acaba em %d\n",problem->task[t].starting_date,problem->task[t].ending_date);

}

}

}

fprintf(fp,"\nForam feitas %d tarefas.\n",nrTasksGeral);

fprintf(fp,"--------------------------------\n");

}

if (optionChosen == 5) {

int *melhorAssignedTo; //Array que guarda a melhor combinação de atribuição de tasks

int **tarefasProgramador; //Array de arrays que guarda as tarefas que cada programador faz numa dada combinação

tarefasProgramador=(int**)malloc(sizeof(int*)*problem->P); //Alocar espaço para o array de arrays 'tarefasProgramador'

for(int i=0;i<problem->P;i++) { tarefasProgramador[i]=(int*)malloc(sizeof(int)*problem->T); }



melhorAssignedTo=(int*)malloc(sizeof(int)*problem->T); //Alocar espaço para o array 'melhorAssignedTo'

for(int i=0;i<problem->T;i++) { melhorAssignedTo[i]=-1; }

int n = problem->T; //Define a variável n com o número de tarefas

int arr[n]; //Inicializa o array 'arr' com n espaços

generateAllBinaryStrings(n, arr, 0, tarefasProgramador, melhorAssignedTo, problem); //Chama a função que gera as combinações binárias



for(int i=0;i<problem->T;i++) { //No final de todas as combinações, copiamos o valor do array melhorAssignedTo para o array assignedTo do problema
```

```
problem->task[i].assigned_to=melhorAssignedT
o[i];

}

int nrTasks=0;

fprintf(fp, "----- Solução a ignorar os
lucros! -----\n");

for(int p=0;p<problem->P;p++) {

fprintf(fp,"\nPROGRAMADOR %d\n",(p+1));

for(int t=0;t<problem->T;t++){

if(problem->task[t].assigned_to==p){

nrTasks++;

fprintf(fp,"Foi atribuída a task que começa
em %d e acaba em
%d\n",problem->task[t].starting_date,problem
->task[t].ending_date);

}

}

}

fprintf(fp,"\nO número total de tarefas
feitas é %d\n\n",nrTasks);

}

}

else {

if (optionChosen == 1) {

int i;
```

```
int nrTasks;

int nrTasksGeral;

int *melhorAssignedTo; //Array que guarda a
melhor combinação de atribuição de tasks

int **combinacoes; //Array de arrays que
guarda todas as combinações binárias (cada
array é uma combinação)

int **tarefasProgramador; //Array de arrays
que guarda as tarefas que cada programador
faz numa dada combinação

combinacoes=(int**)malloc(sizeof(int*)*pow(2
,problem->T)); //Alocar espaço para o array
de arrays 'combinacoes'. O espaço é 2^T
porque existem 2^T combinações

for(int i=0;i<pow(2,problem->T);i++) //visto
que cada task pode ou não ser feita

{

combinacoes[i]=(int*)malloc(sizeof(int)*prob
lem->T);

}



tarefasProgramador=(int**)malloc(sizeof(int*
)*problem->P); //Alocar espaço para o array
de arrays 'tarefasProgramador'

for(int i=0;i<problem->P;i++)

{

tarefasProgramador[i]=(int*)malloc(sizeof(in
t)*problem->T);
```

```
}



melhorAssignedTo=(int*)malloc(sizeof(int)*pr
oblem->T); //Alocar espaço para o array
'melhorAssignedTo'


int n = problem->T; //Define a variável n
com o número de tarefas

int arr[n]; //Inicializa o array 'arr' com n
espaços



for(int i=0;i<pow(2,problem->T);i++)
//Atribui o valor -1 a todas as posições do
array de arrays 'combinacoes'

{

for(int k=0;k<problem->T;k++)

{

combinacoes[i][k]=-1;

}

}

gerarCombinacoes(n, arr, 0,combinacoes);
//Chama a função que gera as combinações
binárias
```

```
for(int i=0;i<problem->T;i++) //Inicializar
o vetor melhorAssignedTo a -1

{

melhorAssignedTo[i]=-1;

}



nrTasksGeral=0; //Define-se a variável
nrTasksGeral (que vai guardar o melhor
número de tasks realizadas possível) a 0

int profitAtual; //Inicializa-se a variável
profitAtual(que vai guardar o profit)

int profitGeral=0;

for(int
comb=0;comb<pow(2,problem->T);comb++) //Para
cada combinação

{



for(int i=0;i<problem->P;i++) //Inicializar
o vetor busy a -1

{

problem->busy[i]=-1;

}



for(int i=0;i<problem->P;i++) //Inicializar
o vetor tarefasProgramador a -1

{

for(int k=0;k<problem->T;k++)

{

tarefasProgramador[i][k]=-1;

}

}



for(int i=0;i<problem->T;i++) //Inicializar
o vetor assignedTo a -1

{

problem->task[i].assigned_to=-1;

}



nrTasks=0;

profitAtual=0;

//printf("Para a combinacao %d\n",comb);

for(int prog=0;prog<problem->P;prog++)
//Para cada programador

{

for(int tar=0;tar<problem->T;tar++)

{

if(combinacoes[comb][tar]==1) //Caso a
tarefa tar da combinacao comb possa ser
feita

{

if(problem->busy[prog]==-1) //Caso o
programador esteja disponível

{

if(problem->task[tar].assigned_to==-1)

{

tarefasProgramador[prog][nrTasks]=tar;

problem->busy[prog]=problem->task[tar].endin
g_date;

problem->task[tar].assigned_to=prog;

nrTasks++;

profitAtual=profitAtual+problem->task[tar].p
rofit;

//printf("A atribuir task %d ao programador
%d\n",tar,prog);

}

}



else //Caso o programador esteja ocupado

{

if(problem->task[tar].assigned_to==-1)
```

```
{

if(problem->busy[prog]<problem->task[tar].st
arting_date)

{

tarefasProgramador[prog][nrTasks]=tar;

problem->busy[prog]=problem->task[tar].endin
g_date;

problem->task[tar].assigned_to=prog;

nrTasks++;

profitAtual=profitAtual+problem->task[tar].p
rofit;

}

}

}

}

}

int flagE=0;

for(int i=0;i<problem->T;i++) //Para cada
elemento da combinacao

{

if(combinacoes[comb][i]==1) //Caso a task
tenha que ser realizada

{

if(problem->task[i].assigned_to==-1) //Caso
a task não tenha sido atribuída

{

//printf("Combinação é inviável\n");

flagE=1;

break;

}

}

}

if(flagE==0) //Caso todas as tasks que
tinham que ser feitas tiverem sido feitas

{

if(profitAtual>profitGeral)

{

for(int i=0;i<problem->T;i++) //Inicializar
o vetor melhorAssignedTo a -1

{

melhorAssignedTo[i]=-1;

}

nrTasksGeral=nrTasks;

profitGeral=profitAtual;

for(int i=0;i<problem->P;i++)

{

for(int k=0;k<problem->T;k++)

{

if(tarefasProgramador[i][k]!=-1) //Caso o
elemento seja uma tarefa

{

melhorAssignedTo[tarefasProgramador[i][k]]=i
;

}

}

}

}

}

}

for(int i=0;i<problem->T;i++) //No final de
todas as combinações, copiamos o valor do
array melhorAssignedTo para o array
assignedTo do problema

{

problem->task[i].assigned_to=melhorAssignedT
o[i];

}
```

```
problem->total_profit=profitGeral; //Atu



fprintf(fp,"Solução com Lucros\n\n");

for(int p=0;p<problem->P;p++)

{

fprintf(fp,"\nPara o Programador
%d\n",(p+1));

for(int t=0;t<problem->T;t++)

{

if(problem->task[t].assigned_to==p)

{

fprintf(fp,"Foi atribuída a task que começa
em %d e acaba em %d com lucro de
%d\n",problem->task[t].starting_date,problem
->task[t].ending_date,problem->task[t].profi
t);

}

}

}

fprintf(fp,"\nForam feitas %d
tarefas.\n",nrTasksGeral);

fprintf(fp,"O profit total é
%d\n",problem->total_profit);

fprintf(fp,"-------------------------------
----\n");

}

if (optionChosen == 2) {

int *melhorAssignedTo; //Array que guarda a
melhor combinação de atribuição de tasks

int **tarefasProgramador; //Array de arrays
que guarda as tarefas que cada programador
faz numa dada combinação

tarefasProgramador=(int**)malloc(sizeof(int*
)*problem->P); //Alocar espaço para o array
de arrays 'tarefasProgramador'

for(int i=0;i<problem->P;i++) {
tarefasProgramador[i]=(int*)malloc(sizeof(in
t)*problem->T); }



melhorAssignedTo=(int*)malloc(sizeof(int)*pr
oblem->T); //Alocar espaço para o array
'melhorAssignedTo'

for(int i=0;i<problem->T;i++) {
melhorAssignedTo[i]=-1; }

int n = problem->T; //Define a variável n
com o número de tarefas

int arr[n]; //Inicializa o array 'arr' com n
espaços

generateAllBinaryStrings(n, arr, 0,
tarefasProgramador, melhorAssignedTo,
problem); //Chama a função que gera as
combinações binárias

for(int i=0;i<problem->T;i++) { //No final
de todas as combinações, copiamos o valor do
array melhorAssignedTo para o array
assignedTo do problema

problem->task[i].assigned_to=melhorAssignedT
o[i];

}

int nrTasks=0;

fprintf(fp, "----- Solução a contabilizar os
lucros! -----\n");

for(int p=0;p<problem->P;p++) {

fprintf(fp,"\nPROGRAMADOR %d\n",(p+1));

for(int t=0;t<problem->T;t++){

if(problem->task[t].assigned_to==p){

fprintf(fp,"Foi atribuída a task que começa
em %d e acaba em %d com lucro de
%d\n",problem->task[t].starting_date,problem
->task[t].ending_date,problem->task[t].profi
t);

}

}

}

fprintf(fp,"\nO profit total é
%d\n\n",problem->total_profit);

}

}

//
```

```
// call your (recursive?) function to solve
the problem here

problem->cpu_time = cpu_time() -
problem->cpu_time;

printf("...Terminou\n");

//

// save solution data

//

fprintf(fp,"\n\n\n--------------------INFORM
AÇÕES DE CONSULTA--------------------\n");

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);

printf("Solution time =
%.3e\n",problem->cpu_time);

fprintf(fp,"%5s %15s %15s %10s" ,"Tasks",
"Starting date", "Ending date","Profit\n");

#define TASK problem->task[i]

for(i = 0;i < problem->T;i++)

fprintf(fp,"%5d %15d %15d %10d\n",i,
TASK.starting_date,TASK.ending_date,TASK.pro
fit);

#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]);

if (I == 1) {

int option;

printf("Você escolheu ignorar os lucros!
Temos 5 implementações que você poderá
escolher!\n(1) SEGUNDA ABORDAGEM\n(2)
TERCEIRA ABORDAGEM\n(3) QUARTA
ABORDAGEM\n(4) QUINTA ABORDAGEM\n(5) SEXTA
ABORDAGEM\nInsira um dos 5 números: \n");

scanf("%d", &option);

if ((option == 1) || (option == 2) ||
(option == 3) || (option == 4) || (option ==
5)){

init_problem(NMec,T,P,I,&problem, option);

solve(&problem, option);

}

else {

printf("Opção inválida!");

return EXIT_FAILURE;

}

}

else if (I == 0) {

int option;

printf("Você escolheu não ignorar os lucros!
Temos 2 implementações que você poderá
```

```
escolher!\n(1) PRIMEIRA ABORDAGEM\n(2)
SEGUNDA ABORDAGEM\nInsira um dos 3 números:
\n");

scanf("%d", &option);

if ((option == 1) || (option == 2)){

init_problem(NMec,T,P,I,&problem, option);

solve(&problem, option);

}

else {

printf("Opção inválida!");

return EXIT_FAILURE;

}

}

else {

printf("O valor do I é inválido! Escolha 1
para ignorar os profits ou 0 para não
ignorar os profits!\n");

}

return 0;

}
```

6.2 Código em Matlab

```
%Código para os gráficos do tempo de execução em função do número de programadores

fidProg=fopen('dianaftnl.txt','r');

TempoProg3=fscanf(fidProg,'%f');

Prog3=[1:10];

plot(Prog3,TempoProg3)

title("Tempo em função dos programadores (para 20 tasks)");

xlabel("Número de programadores");

ylabel("Tempo de execução (s)");

xlim([1,10]);

ylim([0,2]);

fclose(fidProg);
```

```
%Código para os gráficos do tempo de execução em função do número de tarefas

fidProg=fopen('abordagem 5.txt','r');

TempoTarefa5=fscanf(fidProg,'%f');

Tarefas5=[1:24];

plot(Tarefas5,TempoTarefa5)

title("Tempo em função das Tasks Realizadas (para 2 programadores)");

xlabel("Número de tarefas");

ylabel("Tempo de execução (s)");

xlim([1,24]);

ylim([0,]);

fclose(fidProg);
```

```
%Código para os gráficos com informações de mais do que uma abordagem (várias funções representadas num só gráfico)


Tarefas=[1:64];

hold on;

plot(Tarefas,TempoTarefa2,'m')

plot(Tarefas,TempoTarefa3,'r')

plot(Tarefas,TempoTarefa4,'k')

Tarefas=[1:24];

plot(Tarefas,TempoTarefa5,'b')

Tarefas=[1:30];

plot(Tarefas,TempoTarefa6,'g')

hold off;
```

```
title("Tempo em função das Tasks Realizadas (para 2 programadores)");

xlabel("Número de tarefas");

ylabel("Tempo de execução (s)");

xlim([1,64]);

ylim([0,0.00076]);

legend('Implementação 2','Implementação 3','Implementação 4','Implementação 5','Implementação 6','Location','north');
```