Algoritmos e
Estruturas de Dados
Licenciatura em Engenharia Informática
Ano Letivo 2020/2021 2ºAno 1º Semestre

Relatório do Trabalho Prático 1

Generalized weighted job selection problem

Trabalho realizado por:
Diogo Cruz (98595) - 30%
Gonçalo Leal (98008) - 40%
Sophie Pousinho (97814) - 30%

Problema

O exercício proposto pelo professor consistia em encontrar a melhor distribuição de um conjunto de tarefas de programação (T) a serem desenvolvidas por um conjunto de programadores (P), em que a cada tarefa estava associada uma data de início, uma data de término e um lucro, que só seria considerado se a tarefa fosse, de facto, realizada.

Para que uma tarefa pudesse ser atribuída a um qualquer programador, existia um conjunto de considerações e limitações de modo a obter a melhor solução possível. Por um lado, a soma dos lucros das tarefas de programação realizadas devia ser maximizada; por outro lado, uma tarefa apenas podia ser realizada por um único programador, sendo que esse mesmo deveria comprometer-se a realizá-la até ao fim, dessa forma, cada programador só poderia trabalhar numa tarefa de cada vez.

Fontes de materiais externos adaptados na realização deste trabalho:
Slide 135 do material das aulas da UC - Função print_all_sums

Método de Resolução

Para a resolução deste problema, pretendemos seguir a linha de pensamento de paradigmas e técnicas usadas a nível computacional para gerar algoritmos que respondem a um dado problema.
Começámos por analisar os diferentes casos possíveis de solução, numa enumeração sistemática, e depois selecionar o(s) caso(s) que melhor satisfaziam o enunciado, isto é, os casos em que o lucro total da realização de tarefas era maximizado. Para este caso, obtivemos a solução apresentada na versão *job_selection.c*, na qual foi usado um gerador de números pseudo aleatórios para obter combinações de tarefas (as melhores possíveis) e a partir daí atingir valores de lucros também otimizados. Esta primeira versão funciona e permitiu-nos perceber melhor o problema em questão. Ao fazer uma procura exaustiva, de gerar e testar soluções, encontramos um subconjunto das mesmas mais apropriado podendo assim, dividir o problema em problemas menores. Este foi o ponto de partida da segunda versão do código. Contudo, também é necessário ter em conta a questão da eficiência, por isso, na nova versão, *job_selection_v1.c* tivemos como objetivo principal melhorar a eficiência computacional do nosso código sendo que os valores finais, de lucros e tarefas permaneceram os mesmos. Através de uma maior compreensão do problema na primeira versão, permitiu que, nesta segunda, pudéssemos lidar com o problema com outra perspetiva. Apercebemo-nos que não tínhamos de calcular diretamente as combinações das tarefas mas sim considerar e decidir se uma dada tarefa devia ser feita ou não.

A complexidade computacional para os melhor e pior casos de ambos os algoritmos mantém-se, o que muda é a complexidade computacional do caso médio. Outra das mudanças que contribuiu para a maior eficiência do segundo algoritmo foi o uso de funções recursivas em vez de ciclos for.

Método de Resolução (código)

> Cálculo de combinações de tarefas (versão não recursiva)

Explicação: As combinações de tarefas são indicadas através de um código binário (posições iguais a 0 ou a 1) de tamanho igual ao número de tarefas do problem considerado. Inicialmente, nenhuma tarefa foi entregue a nenhum programador, por isso, todas as posições estão a 0. Nas iterações seguintes serão considerados todos os números que é possível criar com T bits em binário.

```
static void calcCombinations(problem_t *problem){
  int n = problem->T;

  // Todas as combinações de x podem ser obtidas transformando todos os números de 0 até x em binário
  char comb[n+1];
  int pos;

  // init
  for (pos = 0; pos < n; pos++){
    comb[pos] = '0';
  }
  comb[n] = '\0';

  for (;;){
    // chamar função de verificação
    calcProfit(problem, comb, n);

    cleanVars(problem);

    for(pos = n-1; pos >= 0 && comb[pos] == '1'; pos--){
      comb[pos] = '0';
    }
    if (pos < 0){
      break;
    }

    comb[pos] = '1';
  }
```

> Cálculo do lucro (versão não recursiva)

Explicação: As combinações de tarefas são indicadas através de um código binário (posições iguais a 0 ou a 1) que foi passado à função como um char array de tamanho igual ao número de tarefas consideradas. Se na posição i do char array estiver um 1, então, a tarefa correspondente deve ser considerada. Por cada uma das combinações é necessário verificar se ela é possível e se há algum programador disponível para a realizar e, se ambas as condições se verificarem, calcula-se o seu lucro. Se o lucro for melhor do que o melhor encontrado até agora, o novo lucro passa a ser o melhor.

```
void calcProfit(problem_t *problem, char comb[], int end){
  int i;
  int possible = 1;

  // uma combinação poderia ser 001001011 (se problem->T fosse 9)
  // este for vai correr a string ao contrário, ou seja, 110100100, porque o último 1 na verdade representa "realizar a tarefa 0"
  for (i = end - 1; i >= 0; i--){

    // se a posição estiver a um então temos de ver se é possível considerar essa tarefa
    if (comb[i] == '1'){
      // ver se há programadores disponíveis

      int tasks_i = end - (i + 1);
      int p = 0;
      for (p = 0; p < problem->P; p++){
        // se houver algum programador disponível então sai do ciclo
        // o busy guarda o dia em que a tarefa que o programador está a realizar termina, se esse dia for menor que o dia em que a
        // tarefa tasks_i se inicia, então ele está disponível para a relaizar
        if (problem->busy[p] < problem->task[tasks_i].starting_date){
          break;
        }
      }
```

```
      // Há um programador disponível ou o ciclo acabou naturalmente, neste caso p = problem->P
      if (p < problem->P){
        // o programador vai estar ocupado até ao último dia da tarefa
        problem->busy[p] = problem->task[tasks_i].ending_date;
        problem->total_profit += problem->task[tasks_i].profit;
      }
      else{
        possible = 0;
      }
    }

    if (possible == 0){
      break;
    }
  }
```

```
  // como o ciclo corre do fim para o princípio então se i for menor que 0 quer dizer que correu a combinação toda, ou seja,
  // a combinação é possível
  if (i < 0){
    // é possível
    if (problem->total_profit > problem->best_profit){
      // if true change best_profit, best_distribution and tasks_combination
      problem->best_profit = problem->total_profit;
      for (int i = 0; i < MAX_P; i++){
        problem->best_distribution[i] = problem->busy[i];
      }
      for (int i = 0; i < MAX_T; i++){
        problem->tasks_combination[i] = comb[i];
      }
    }

    problem->total_possible++;
  }
}
```

> Atribuição de uma tarefa (versão recursiva)

Explicação: Para uma tarefa ser atribuída é necessário verificar , entre outras coisas, se há algum programador disponível. Se houver, o valor desse programador no busy array passa a ser a data de término dessa tarefa (só poderá ser atribuída outra tarefa a esse programador se a data de início for superior à data final da tarefa atual). Para além disso, o lucro total acresce ao valor de lucro da tarefa que acabou de ser atribuída.

```
static int assignTasks(problem_t *problem, int tasks2assign[], int n_tasks, int tasks_i){
  int p;

  // se o tasks_i for igual ao n_tasks então já correu a combinação toda e esta é possível
  if(tasks_i == n_tasks){
    if (problem->total_profit > problem->best_profit){

      problem->best_profit = problem->total_profit;
      for (int i = 0; i < MAX_P; i++){
        problem->best_distribution[i] = problem->busy[i];
      }
      for (int i = 0; i < n_tasks; i++){
        problem->tasks_combination[i] = tasks2assign[i];
      }
    }
    return 1;
  }
```

```
  // se for zero não é para considerar, então avança
  if (tasks2assign[tasks_i] == 0){
    assignTasks(problem, tasks2assign, n_tasks, tasks_i+1);
  }
  else{

    for (p = 0; p < problem->P; p++){
      if (problem->busy[p] < problem->task[tasks_i].starting_date){
        break;
      }
    }

    if (p < problem->P){
      // há um programador disponível
      problem->busy[p] = problem->task[tasks_i].ending_date;
      problem->total_profit += problem->task[tasks_i].profit;

      assignTasks(problem, tasks2assign, n_tasks, tasks_i+1);
    }

    else{
      return 0;
    }
  }
}
```

> Cálculo do melhor lucro (versão recursiva)

Explicação: Esta função foi inspirada na função print_all_sums que está nos slides da UC. Esta função consiste em percorrer dois caminhos distintos para cada tarefa: considerar a tarefa ou não considerar a tarefa. Quando a posição do array onde vamos escrever for igual ao número de tarefas quer dizer que já chegamos ao fim da combinação e podemos verificar se esta é possível.

```c
static void calcBestProfit(problem_t *problem, int i, int arr[]){
  int n = problem->T;

  if (i == n){
    if (assignTasks(problem, arr, n, 0)) problem->total_possible = problem->total_possible + 1;
    // we must clean busy and total profit after every call to assign tasks function
    cleanVars(problem);

    return;
  }
  else{

    // we have to store the array of tasks and the position to be able to return to this point
    // after doing the path of not considering the task i
    int i_backup = i;
    int arr_backup[n];

    for (int j = 0; j < n; j++){
      arr_backup[j] = arr[j];
    }

    calcBestProfit(problem, ++i, arr); // don't consider task i

    // restore variables
    i = i_backup;
    // consider task i
    arr_backup[i] = 1;

    calcBestProfit(problem, ++i, arr_backup); // consider task i
  }
}
```

> Limpeza do array dos programadores ocupados (versão recursiva e não recursiva)

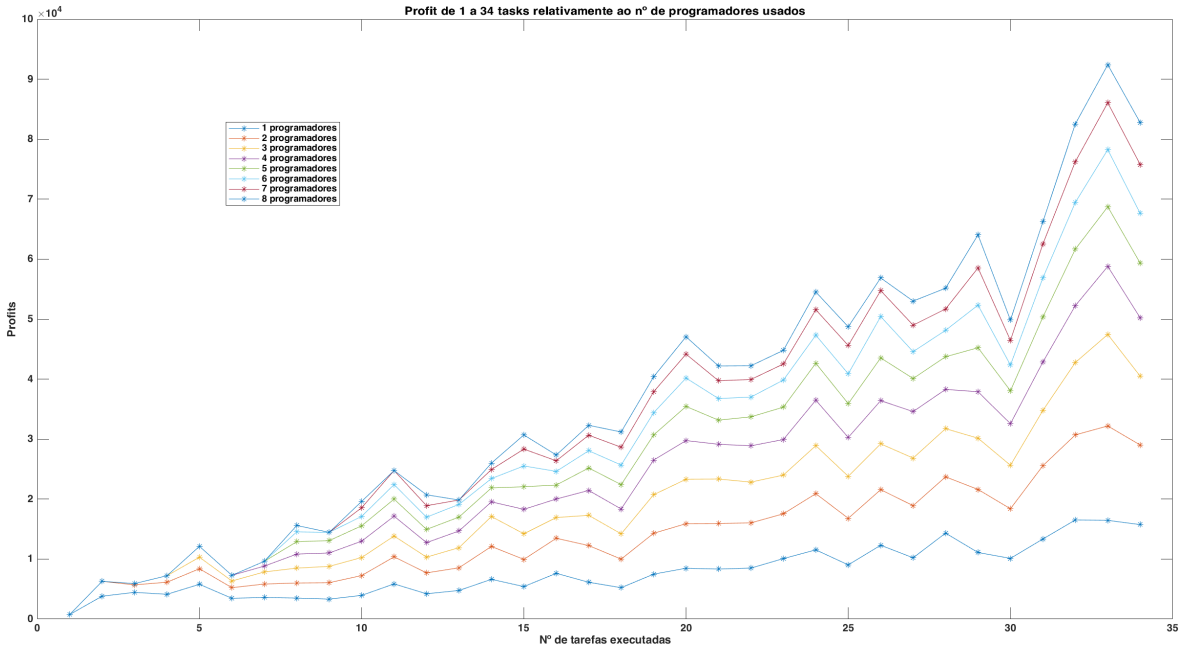
Explicação: Os valores no array devem ser limpos de cada vez que é feita uma nova combinação de tarefas uma vez que diferentes combinações geram diferentes atribuições aos programadores. Caso contrário, sempre que fossemos considerar uma combinação nova corríamos o risco de ter todos os programadores erradamente ocupados. É, também, necessário igualar o lucro atual a 0, porque o lucro é acrescentado, por isso, se esta variável não for limpa obteremos resultados errados.

```
void cleanVars(problem_t *problem){
  // se não limparmos o array do busy quando formos verificar outra combinação os programadores vão estar
  // erradamente ocupados com as tarefas da combinação anterior
  for (int i = 0; i < problem->P; i++){
    problem->busy[i] = -1;
  }

  // limpar o lucro atual da combinação, para cada uma das combinações tem de começar a zero
  problem->total_profit = 0;
}
```

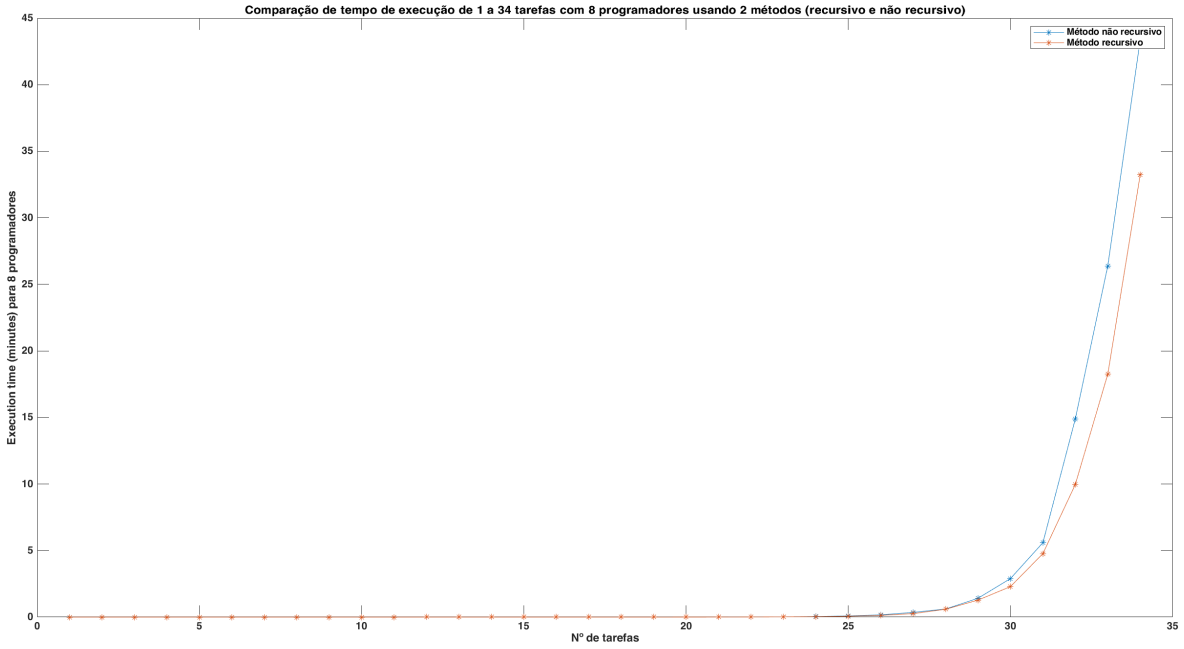
Gráficos e Análise de Resultados

> Gráfico dos ***Profits*** na execução de 1 a 34 tasks, executadas de 1 a 8 programadores em simultâneo



Pelo gráfico podemos concluir que para o aumento do número de programadores (uma linha do gráfico por programador como é mostrado pelas cores na legenda do gráfico) na realização das tasks, a execução quer seja de 5 tasks ou de 34, o profit vai ser sempre maior consoante maior for o número de programadores a trabalhar nas dadas tarefas.

> Gráfico do **tempo de execução** de 1 a 34 tasks executadas por 8 programadores em ambos os métodos utilizados na resolução deste projeto.



Como percebemos pelo gráfico, para 8 programadores, o tempo de execução entre o modo recursivo e não recursivo torna-se bastante significativo à medida que o número de tarefas a serem executadas aumenta.
Concluímos que quanto maior for o número de tarefas a serem realizadas, maior vai ser a eficácia do método recursivo comparativamente ao método não recursivo.
Vemos no exemplo aqui que para 34 tarefas o método recursivo aumenta a eficácia em relação ao não recursivo por mais de 10 minutos.

Conclusão

As soluções encontradas pelo nosso grupo não foram as mais eficientes nem as mais otimizadas. No entanto, pensamos ter cumprido o objetivo do trabalho tendo conseguido criar um algoritmo capaz de resolver o problema apresentado. Como sugerido pelo docente, começámos por implementar uma solução *brute force* em que não era tido em conta o tempo de execução. Depois, já com uma melhor compreensão do que tinha que ser feito, conseguimos melhorar o algoritmo usando funções recursivas e com ligeiras alterações de raciocínio.

Uma forma de aumentar a eficiência do nosso algoritmo seria ir verificando a cada chamada da função recursiva se aquela combinação de tarefas era possível. Tentamos implementar esta lógica, mas não tivemos sucesso. Parte desse insucesso deve-se ao nosso pouco à vontade com funções recursivas. Para nós este é ainda um conceito pouco explorado no curso.

Outra das dificuldades que tivemos foi comparar tempos de execução com outros colegas, por estarmos a usar uma máquina virtual e por termos computadores com piores características.

Em suma, este trabalho, tal como esta UC, foram uma grande vantagem no nosso desenvolvimento enquanto programadores. Quer através do desenvolvimento de conceitos previamente aprendidos, mas pouco explorados, quer pela aprendizagem de novos conceitos ou técnicas algorítmicas.

Anexos

Solução base: job_selection.c

```c
////////////////////////////////////////////////////////////////////////////////////////////////////
//
// problem solution (place your solution here)

void cleanVars(problem_t *problem){
  // se não limparmos o array do busy quando formos verificar outra combinação os programadores vão estar erradamente ocupados com as tarefas da combinação anterior
  for (int i = 0; i < problem->P; i++){
    problem->busy[i] = -1;
  }

  // limpar o lucro atual da combinação, para cada uma das combinações tem de começar a zero
  problem->total_profit = 0;
}

void calcProfit(problem_t *problem, char comb[], int end){
  int i;
  int possible = 1;

  // uma combinação poderia ser 001001011 (se problem->T fosse 9)
  // este for vai correr a string ao contrário, ou seja, 110100100, porque o último 1 na verdade representa "realizar a tarefa 0"
  for (i = end - 1; i >= 0; i--){

    // se a posição estiver a um então temos de ver se é possível considerar essa tarefa
    if (comb[i] == '1'){
      // ver se há programadores disponíveis

      int tasks_i = end - (i + 1);
      int p = 0;
      for (p = 0; p < problem->P; p++){
        // se houver algum programador disponível então sai do ciclo
        // o busy guarda o dia em que a tarefa que o programador está a realizar termina, se esse dia for menor que o dia em que a
        // tarefa tasks_i se inicia, então ele está disponível para a relaizar
        if (problem->busy[p] < problem->task[tasks_i].starting_date){
          break;
        }
      }

      // Há um programador disponível ou o ciclo acabou naturalmente, neste caso p = problem->P
      if (p < problem->P){
        // o programador vai estar ocupado até ao último dia da tarefa
        problem->busy[p] = problem->task[tasks_i].ending_date;
        problem->total_profit += problem->task[tasks_i].profit;
      }
      else{
        possible = 0;
      }
    }

    if (possible == 0){
      break;
    }
  }

  // como o ciclo corre do fim para o princípio então se i for menor que 0 quer dizer que correu a combinação toda, ou seja,
  // a combinação é possível
  if (i < 0){
    // é possível
    if (problem->total_profit > problem->best_profit){
      // if true change best_profit, best_distribution and tasks_combination
      problem->best_profit = problem->total_profit;
      for (int i = 0; i < MAX_P; i++){
        problem->best_distribution[i] = problem->busy[i];
      }
      for (int i = 0; i < MAX_T; i++){
        problem->tasks_combination[i] = comb[i];
      }
    }

    problem->total_possible++;
  }
}

static void calcCombinations(problem_t *problem){
  int n = problem->T;

  // Todas as combinações de x podem ser obtidas transformando todos os números de 0 até x em binário
  char comb[n+1];
  int pos;

  // init
  for (pos = 0; pos < n; pos++){
    comb[pos] = '0';
  }
  comb[n] = '\0';

  for (;;){
    // chamar função de verificação
    calcProfit(problem, comb, n);

    cleanVars(problem);

    for(pos = n-1; pos >= 0 && comb[pos] == '1'; pos--){
      comb[pos] = '0';
    }
```

```
  comb[n] = '\0';

  for (;;){
    // chamar função de verificação
    calcProfit(problem, comb, n);

    cleanVars(problem);

    for(pos = n-1; pos >= 0 && comb[pos] == '1'; pos--){
      comb[pos] = '0';
    }
    if (pos < 0){
      break;
    }

    comb[pos] = '1';
  }

  printf("----------------------end----------------------\n");
  printf("T = %d; P = %d\n", problem->T, problem->P);
  printf("Best Profit = %d\n", problem->best_profit);
}

//

#if 1

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

  //
  // open log file
  //
  (void)mkdir(problem->dir_name,S_IRUSR | S_IWUSR | S_IXUSR);
  fp = fopen(problem->file_name,"w");
  if(fp == NULL)
  {
    fprintf(stderr,"Unable to create file %s (maybe it already exists? If so, delete it!)\n",problem->file_name);
    exit(1);
  }
```

```
  // solve
  //
 problem->cpu_time = cpu_time();
   // temos de inicializar estas variáveis
   problem->total_profit = 0;
   problem->best_profit = 0;
   problem->total_possible = 0;

   // initialize tasks_combination
   for (int j = 0; j < MAX_T; j++){
     problem->tasks_combination[j] = -1;
   }

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

   // call your (recursive?) function to solve the problem here
   calcCombinations(problem);
 problem->cpu_time = cpu_time() - problem->cpu_time;
 //
 // save solution data
 //
 fprintf(fp,"NMec = %d\n",problem->NMec);
 fprintf(fp,"T = %d\n",problem->T);
 fprintf(fp,"P = %d\n",problem->P);
 fprintf(fp,"Profits%s ignored\n",(problem->I == 0) ? " not" : "");
 fprintf(fp,"Solution time = %.3e\n",problem->cpu_time);
 fprintf(fp,"Task data\n");
#define TASK  problem->task[i]
 fprintf(fp,"Index Start End Profit\n");
 for(i = 0;i < problem->T;i++)
   fprintf(fp,"%5d  %5d %3d %6d\n",i, TASK.starting_date, TASK.ending_date, TASK.profit);
#undef TASK
 if (problem->I == 1){
   fprintf(fp,"Highest Number of Tasks = %d\n",problem->best_profit);
 }
 else{
   fprintf(fp,"Highest Profit = %d\n",problem->best_profit);
 }
 fprintf(fp,"Tasks considered: {");
 for (int j = 0; j < problem->T; j++){
   if (problem->tasks_combination[j] == '1'){
     fprintf(fp,"%d,", problem->T-(j+1));
   }
 }
```

```
  fprintf(fp,"}\n");
  fprintf(fp, "Viable combinations: %d\n", problem->total_possible);
  fprintf(fp,"End\n");
  //
  // terminate
  //
  if(fflush(fp) != 0 || ferror(fp) != 0 || fclose(fp) != 0)
  {
    fprintf(stderr,"Error while writing data to file %s\n",problem->file_name);
    exit(1);
  }
}

#endif



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

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

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

Solução otimizada do código base: job_selecton_v1.c

```c
////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// problem solution (place your solution here)

void cleanVars(problem_t *problem){
  // se não limparmos o array do busy quando formos verificar outra combinação os programadores vão estar
  // erradamente ocupados com as tarefas da combinação anterior
  for (int i = 0; i < problem->P; i++){
    problem->busy[i] = -1;
  }

  // limpar o lucro atual da combinação, para cada uma das combinações tem de começar a zero
  problem->total_profit = 0;
}

// problem_t *problem -> pointer to problem info
// int tasks2assign   -> is our combination o r tasks from taskCombination
// int n_tasks        -> same as length(tasks2assign)
// int tasks_i        -> index where we are at tasks2assign[]
// int profit         -> profit until now with this combination
// int final_comb     -> always 0 in the first version, but will be used in version 2.0 ;)
static int assignTasks(problem_t *problem, int tasks2assign[], int n_tasks, int tasks_i){
  int p;

  // se o tasks_i for igual ao n_tasks então já correu a combinação toda e esta é possivel
  if(tasks_i == n_tasks){
    if (problem->total_profit > problem->best_profit){

      problem->best_profit = problem->total_profit;
      for (int i = 0; i < MAX_P; i++){
        problem->best_distribution[i] = problem->busy[i];
      }
      for (int i = 0; i < n_tasks; i++){
        problem->tasks_combination[i] = tasks2assign[i];
      }
    }
    return 1;
  }

  // se for zero não é para considerar, então avança
  if (tasks2assign[tasks_i] == 0){
    assignTasks(problem, tasks2assign, n_tasks, tasks_i+1);
  }
  else{
```

```c
    for (p = 0; p < problem->P; p++){
      if (problem->busy[p] < problem->task[tasks_i].starting_date){
        break;
      }
    }

    if (p < problem->P){
      // há um programador disponivel
      problem->busy[p] = problem->task[tasks_i].ending_date;
      problem->total_profit += problem->task[tasks_i].profit;

      assignTasks(problem, tasks2assign, n_tasks, tasks_i+1);
    }

    else{
      return 0;
    }
  }
}

// problem_t *problem -> pointer to problem info
// int i              -> represents the task we are considering or not
// int arr[]          -> is our array of tasks, if arr[x] = 1 then we are considering task x
static void calcBestProfit(problem_t *problem, int i, int arr[]){
  int n = problem->T;

  if (i == n){
    if (assignTasks(problem, arr, n, 0)) problem->total_possible = problem->total_possible + 1;
    // we must clean busy and total profit after every call to assign tasks function
    cleanVars(problem);

    return;
  }
  else{

    // we have to store the array of tasks and the position to be able to return to this point
    // after doing the path of not considering the task i
    int i_backup = i;
    int arr_backup[n];

    for (int j = 0; j < n; j++){
      arr_backup[j] = arr[j];
    }

    calcBestProfit(problem, ++i, arr); // don't consider task i
```

```
    // restore variables
    i = i_backup;
    // consider task i
    arr_backup[i] = 1;

    calcBestProfit(problem, ++i, arr_backup); // consider task i
  }
}

//

#if 1

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

  //
  // open log file
  //
  (void)mkdir(problem->dir_name,S_IRUSR | S_IWUSR | S_IXUSR);
  fp = fopen(problem->file_name,"w");
  if(fp == NULL)
  {
    fprintf(stderr,"Unable to create file %s (maybe it already exists? If so, delete it!)\n",problem->file_name);
    exit(1);
  }
  //
  // solve
  //
  problem->cpu_time = cpu_time();
    problem->best_profit = 0;
    problem->total_possible = 0;
    // initialize tasks_combination
    for (int j = 0; j < MAX_T; j++){
      problem->tasks_combination[j] = -1;
    }
    for (int j = 0; j < MAX_P; j++){
      problem->busy[j] = -1;
    }
    problem->total_profit = 0;

    // call your (recursive?) function to solve the problem here
    int arr[problem->T];
    for (int j = 0; j < problem->T; j++){
      arr[j] = 0;
    }
```

```
    calcBestProfit(problem, 0, arr);
    printf("-----------------------end-----------------------\n");
    printf("T = %d; P = %d\n", problem->T, problem->P);
    printf("Best Profit = %d\n", problem->best_profit);
    printf("Viable combinations = %d\n", problem->total_possible);
  problem->cpu_time = cpu_time() - problem->cpu_time;
  //
  // save solution data
  //
  fprintf(fp,"NMec = %d\n",problem->NMec);
  fprintf(fp,"T = %d\n",problem->T);
  fprintf(fp,"P = %d\n",problem->P);
  fprintf(fp,"Profits%s ignored\n",(problem->I == 0) ? " not" : "");
  fprintf(fp,"Solution time = %.3e\n",problem->cpu_time);
  fprintf(fp,"Task data\n");
#define TASK  problem->task[i]
  fprintf(fp,"Index Start End Profit\n");
  for(i = 0;i < problem->T;i++)
    fprintf(fp,"%5d  %5d %3d %6d\n",i, TASK.starting_date, TASK.ending_date, TASK.profit);
#undef TASK
  if (problem->I == 1){
    fprintf(fp,"Highest Number of Tasks = %d\n",problem->best_profit);
  }
  else{
    fprintf(fp,"Highest Profit = %d\n",problem->best_profit);
  }
  fprintf(fp,"Tasks considered: {");
  for (int j = 0; j < problem->T; j++){
    if (problem->tasks_combination[j] != 0){
      fprintf(fp,"%d,", j);
    }
  }
  fprintf(fp,"}\n");
  fprintf(fp, "Viable combinations: %d\n", problem->total_possible);
  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
```

```c
////////////////////////////////////////////////////////////////////////////////////////////////////
//
// main program
//

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

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

Código bash de processamento dos resultados: run_v1.sh

```bash
#!/bin/bash

for (( i=1; i <= $2; i++ ));
do
    for ((j = 1; j <= $3; j++));
    do
        ./job_selection_v1 $1 $i $j $4
    done
done

for (( i=1; i <= $2; i++ ));
do
    for ((j = 1; j <= $3; j++));
    do
        ./job_selection $1 $i $j $4
    done
done

# obter lucros e tempos de cada ficheiro
cd 098008;

if [ -f solution_times.txt ];
then
    rm solution_times.txt;
fi

if [ -f highest_profits.txt ];
then
    rm highest_profits.txt;
fi

for (( i=1; i <= $2; i++ ));
do
    printf "%d " $i >> solution_times.txt;
    printf "%d " $i >> highest_profits.txt;

    for ((j = 1; j <= $3; j++));
    do
        file=$(printf "%02d_%02d_%d.txt" $i $j $4);
        if [ -f "$file" ];
        then
            solution_time=$(grep "Solution time = " $file | awk '{print $4}');
            highest_profit=$(grep "Highest Profit = " $file | awk '{print $4}');

            printf "%s" $solution_time >> solution_times.txt;
            printf "%s" $highest_profit >> highest_profits.txt;
        fi

        printf " ; " >> solution_times.txt;
        printf " ; " >> highest_profits.txt;
    done

    printf "\n" >> solution_times.txt;
    printf "\n" >> highest_profits.txt;
done

cd ..;

# obter lucros e tempos de cada ficheiro
cd 098008_v1;

if [ -f solution_times.txt ];
then
    rm solution_times.txt;
fi

if [ -f highest_profits.txt ];
then
    rm highest_profits.txt;
fi

for (( i=1; i <= $2; i++ ));
do
    printf "%d " $i >> solution_times.txt;
    printf "%d " $i >> highest_profits.txt;

    for ((j = 1; j <= $3; j++));
    do
        file=$(printf "%02d_%02d_%d.txt" $i $j $4);
        if [ -f "$file" ];
        then
            solution_time=$(grep "Solution time = " $file | awk '{print $4}');
            highest_profit=$(grep "Highest Profit = " $file | awk '{print $4}');

            printf "%s" $solution_time >> solution_times.txt;
            printf "%s" $highest_profit >> highest_profits.txt;
        fi

        printf " ; " >> solution_times.txt;
        printf " ; " >> highest_profits.txt;
    done

    printf "\n" >> solution_times.txt;
    printf "\n" >> highest_profits.txt;
done
```

Código no MATLAB para execução e visualização dos gráficos: graficos.m

```
cd 098008
file_format = "%d %d ; %d ; %d ; %d ; %d ; %d ; %d ;";
fp = fopen('highest_profits.txt','r');
size_times = [9 Inf];
profits = fscanf(fp,file_format,size_times);
%disp(profits);

for k = 2 : 9   %  por programador
    figure(101);
    title("Profit de 1 a 34 tasks relativamente ao nº de programadores usados");
    ylabel("Profits");
    xlabel("Nº de tarefas executadas");
    plot([1:34], profits(k,:), '-*');
    hold on;
end
hold off;

N = 8;
Legend=cell(N,1);
for iter=1 : N
Legend{iter}=strcat(num2str(iter), ' programadores');
end
legend(Legend)


file_format = "%d %f ; %f ; %f ; %f ; %f ; %f ; %f ;";
fp = fopen('solution_times.txt','r');
size_times = [9 Inf];
times = fscanf(fp,file_format,size_times);
%disp(times);
x = times(9,:)/60;
```

```
%%-------------------------------------------------------------------------------------------

cd ..
cd 098008_v1

file_format = "%d %f ; %f ; %f ; %f ; %f ; %f ; %f ; %f ;";
fp = fopen('solution_times.txt','r');
size_times = [9 Inf];
times2 = fscanf(fp,file_format,size_times);
%disp(times);
y = times2(9,:)/60;

figure(305);
plot([1:34], times(9,:)/60, '-*');
hold on;
plot([1:34], times2(9,:)/60, '-*');
title("Comparação de tempo de execução de 1 a 34 tarefas com 8 programadores usando 2 métodos (recursivo e não recursivo)");
legend("Método não recursivo","Método recursivo");
ylabel("Execution time (minutes) para 8 programadores");
xlabel("Nº de tarefas");
hold off;

fclose(fp);
```