



Algoritmos e Estruturas de Dados

Job Selection

Professores:

Tomás Oliveira e Silva (tos@ua.pt)

Pedro Lavrador (plavrador@ua.pt)

Pedro Sobral, 98491 – 45.0%

André Freixo, 98495 – 27.5%

Marta Fradique, 98626 – 27.5%

Índice

1 – Introdução.....	3
2 - Introdução ao Problema.....	4
2.1 - Resumo.....	4
2.2 – Compilação e Execução	4
3 – Resolução do Problema	6
3.1 - Inicialização	6
3.2 – Função Recursiva.....	6
3.3 – Algoritmos	10
3.4 – Implementação em Java do problema	11
4 - Resultados	12
4.1 - Tabelas	12
4.2 – Gráficos.....	16
4.2.1 – Profit.....	16
4.2.2 – Tempos de execução	18
4.2.3 – Ignorar Profits.....	20
5-Apêndice	24
5.1 -Tabelas	24
5.2 - Matlab	24
5.3- Ficheiros Bash	28
5.4 – Criação de novas variáveis	28
5.5 – Função recursiva.....	29
6 - Conclusão.....	30
7 - Bibliografia.....	31

1 – Introdução

No âmbito da unidade curricular de AED, foi-nos apresentada a realização deste trabalho prático, sendo este relatório o resultado do código “Job Selection Problem”. Todo o código fonte e informações deste trabalho prático podem ser encontradas neste [repositório do GitHub](#)¹ (mais informações, ler README.md do repositório).

Dado um número de tarefas, e um número de programadores, o programa implementado em C, de forma genérica, através de um algoritmo recursivo feito por nós, retorna a melhor maneira de realização do trabalho, ou seja, a maneira em que se consegue obter mais lucro, sem que ocorra sobreposição de programadores.

A linguagem de programação C, é uma linguagem muito poderosa, pois dá ao programador um controlo íntegro de todo o processo programado, sendo uma linguagem onde o programador tem de lidar com todos os pormenores, torna-se consideravelmente eficiente e otimizada. **O algoritmo principal por nós desenvolvido, irá usar a técnica da recursividade, pois dividirmos o problema principal em subproblemas é uma maneira eficiente de chegar a uma resolução viável.**

Para a resolução do problema imposto, irá ser elaborado código C `job_selection.c`, pois trata-se de uma linguagem com tempos de execução extraordinariamente baixos comparado a outras linguagens, por exemplo, Java e Python. Os resultados obtidos serão guardados num ficheiro `.txt` dentro de uma pasta com o `nMec`. Para a análise dos resultados, iremos usar o Matlab, e o nosso conhecimento adquirido ao longo do semestre noutras cadeiras em que o uso desta ferramenta é primordial. Esperamos também criar um script em Shell, para correr o programa `job_selection.c`, com o objetivo de conseguir utilizar todos os cores do processador.

Com a realização deste trabalho prático, esperamos veemente alargar os nossos conhecimentos em C, e principalmente em implementar algoritmos eficientes, e otimizados, para que possamos resolver os problemas propostos de maneira otimizada. Esperamos também conseguir concluir com êxito todos os objetivos que são propostos no início (em comentário) do programa `job_selection.c`. Para uma abordagem mais eficaz está em base de discussão a implementação da resolução do problema também em Java, no entanto, por motivos de tempo e experiência a programar não sabemos se conseguiremos alcançar este objetivo ambicioso.

¹Por motivos de privacidade o repositório encontra-se privado, para visualização, é favor entrar em contacto com os autores do trabalho prático.

2 - Introdução ao Problema

2.1 - Resumo

Sendo direto e conciso, o que se pretende obter é a maneira mais eficiente, quer computacionalmente, quer a nível de *profit*, para as *Tarefas* e para os *Programadores* dados, ou seja, através do algoritmo recursivo implementado por nós, iremos percorrer todas as possibilidades viáveis, através de uma pesquisa exaustiva das mesmas.

Em abstrato, a função recursiva, irá fazer uma pesquisa exaustiva, com o objetivo de encontrar o melhor *profit*, através de um caminho viável, ou seja, um programador só pode realizar uma tarefa de cada vez. Podemos imaginar as possibilidades todas implementadas numa árvore binária, com n níveis, onde n serão o número de *Tarefas*, a função recursiva vai “podar” essa mesma árvore, pois quando encontra uma solução com sobreposição de *Tarefas* para os *Programadores*, esse ramo sai da árvore, reduzindo desta forma o número total de possibilidades e aumentando em larga escala o tempo de execução. À medida que a árvore vai sendo criada e podada, o *profit* vai sendo comparado com o melhor atual, entre outras comparações que serão mencionadas mais à frente.

2.2 – Compilação e Execução

Para compilar (Fig.1) o programa é necessário à partida ter um compilador de C instalado na máquina, por exemplo o *gcc*.

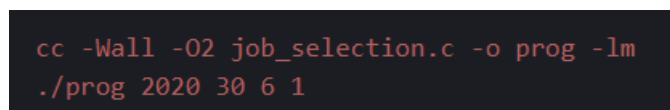
Posto isto, para compilar basta:

```
cc -Wall -O2 job_selection.c -o prog
```

Para correr a compilação basta:

```
./prog NMec T P I
```

Em que o NMec é o número mecanográfico do aluno, o T é o número de Tarefas a realizar, o P o número de Programadores, e o I o ignore_profit que pode tomar valores de um ou 0.



```
cc -Wall -O2 job_selection.c -o prog -lm
./prog 2020 30 6 1
```

Figura 1 – Compilação e execução do código

De modo a aumentar a automação da execução do programa, foi usado o *script job_selection_do_all.bash*(Fig.2), fornecido na página online da unidade curricular, onde através da implementação de ciclos *for*, criamos as possibilidades combinatórias entre *Tarefas* e *Programadores*. Com o objetivo de conseguir diminuir os tempos totais de execução, o *script* também está implementado de modo a que se consiga correr o problema em n terminais,

onde n será o número de núcleos do processador. Por exemplo, numa máquina com um processador com 4 núcleos, conseguimos correr ao mesmo tempo 4 *soluções*.

```

1  #!/bin/bash
2
3  NMec=98491
4  I=0
5  d=$(printf "%06d" $NMec)
6  if [ -d $d ]; then
7      d=$(printf "%06d" $NMec)
8      rm -vf $(grep -L End $d/*)
9  fi
10 for T in {1..40}; do
11     for P in {1..8}; do
12         if (( $T >= $P )); then
13             f=$(printf "%06d/%02d_%02d_%d.txt" $NMec $T $P $I)
14             if [ ! -e $f ]; then
15                 echo $f
16                 ./prog $NMec $T $P $I
17             fi
18         fi
19     done
20 done

```

Figura 2 – *job_selection_do_all_bash*

Nota: para que seja possível correr o *script job_selection_do_all.bash*, é necessário dar permissões ao utilizador, de seguida já há condições para se executar o *script*, (Fig.3).

```

chmod u+x job_selection_do_all.bash
./job_selection_do_all.bash

```

Figura 3 - Código para dar permissão e correr o *job_selection_do_all.bash*

3 – Resolução do Problema

3.1 - Inicialização

A primeira ideia de resolução do problema proposto foi uma função recursiva, no entanto muito “verde”, quer por alguma inexperiência dos membros do respetivo grupo, quer também, mesmo que agora não aparente, pela complexidade do problema, e um pouco também à não familiarização com o C, no início da resolução deste mesmo trabalho.

Ora a ideia inicial começou a ganhar forma e efeito, e desse modo começamos a solucionar alguns pequenos problemas que tínhamos. Posto isto, e de modo geral a arquitetura do nosso algoritmo/função teria de ser: tratar de 2 casos especiais/exceções, o início, para a inicialização de determinadas variáveis, e o fim (do ramo da árvore), onde se irão fazer certas e determinadas comparações relativamente ao profit. No restante código da nossa implementação será então feita a execução das duas e possibilidades, incluir a tarefa ou não incluir a tarefa.

3.2 – Função Recursiva

A nossa função recursiva é dotada de dois argumentos à cabeça (Fig.4), um deles o problema que a mesma irá tratar, o outro onde irá começar.

```
264 void recursive_function(problem_t *problem, int i) {
```

Figura 4 – Início função recursiva

À partida, a nossa função tem duas exceções, uma para quando o *i* é igual a zero, e outra para quando o *i* é igual ao número de tarefas do problema em questão (quando o ramo acaba). Quando começa (Fig.5), inicializamos as variáveis *total_profit*, *best_total_profit* e *terminal_cases* com os valores 0 (zero), e através de um ciclo *for*, inicializamos o array *busy* todo a -1 (menos um), onde -1, significa o programador estar livre.

```
357 {
358     problem->total_profit = 0;
359     problem->best_total_profit = 0;
360     problem->terminal_cases = 0;
361     //Inicializar busy a '-1'
362     for (int k = 0; k < problem->P; k++) {
363         problem->busy[k] = -1;
364     }
365 }
366
367 recursive_function(problem, 0);
```

Figura 5 – Inicialização de variáveis

Esta era a nossa primeira implementação, porém posteriormente, numa análise mais cuidada e seletiva, vimos que a condição de comparação do i igual a 0, era verificada todas as vezes que a função era chamada, sendo que só na primeira chamada da mesma é que o bloco de código dentro da condição é que era executado. Deste modo, decidi-mos inicializar estas variáveis antes da chamada da função recursiva no código. Depois de alguns testes, e registo dos tempos de execução de uma *task* escolhida arbitrariamente, contruímos o seguinte gráfico(Fig.6), que de uma maneira substancial mostra-nos que se consegue tornar toda a estrutura mais eficiente, eficiência essa que ronda a casa de um par de décimas no pior dos casos, e chega a 1.5 segundos no melhor deles.

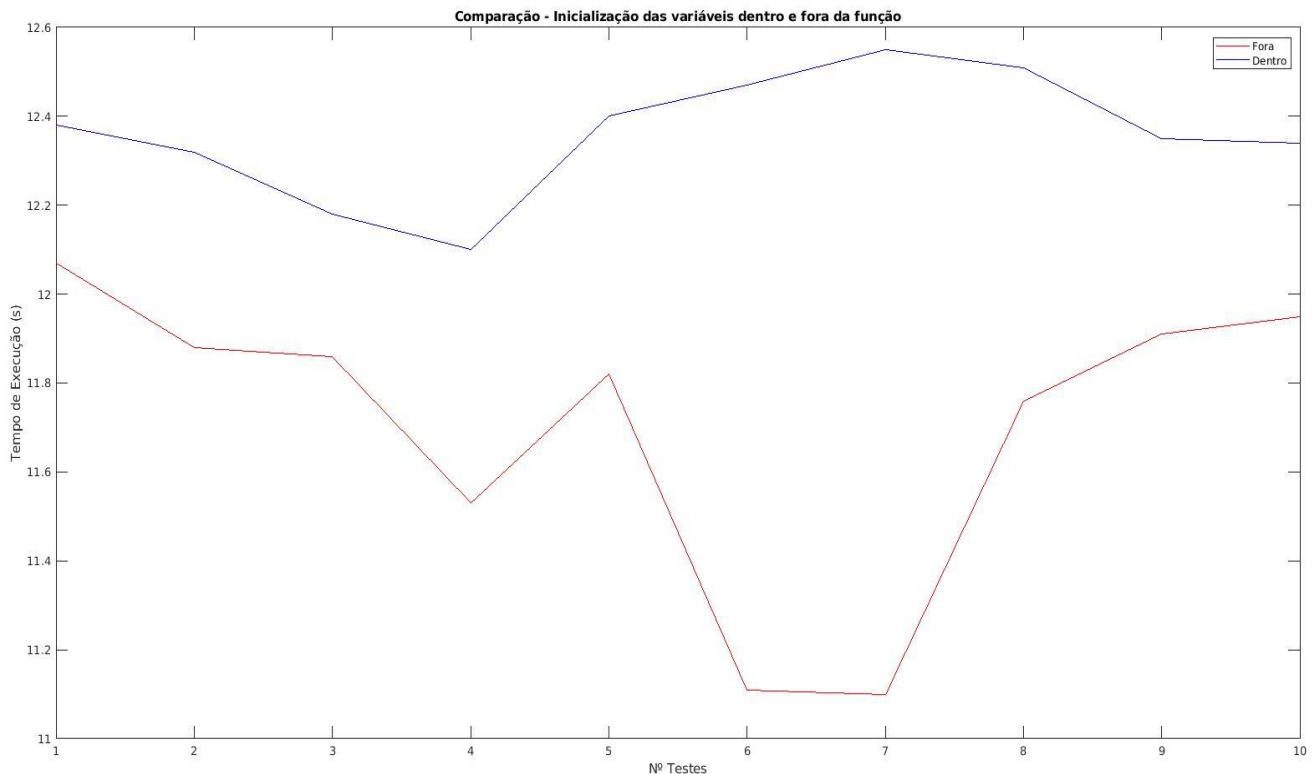


Figura 6 - Comparação tempos de execução

No seguimento da introdução deste ponto do relatório e relativamente ao caso especial de quando acaba, ou seja, o i é igual ao número de *Tarefas* (Fig.7), fazemos a comparação do *total_profit* com o *best_total_profit*, e caso o primeiro seja maior que o segundo, o *best_total_profit* fica atualizado com o valor do *total_profit*, a seguir inicializamos a variável *number_solutions* com 1, que irá servir como contador do número de soluções para quando o *profit* é ingorado. Ainda nesta secção é atualizado o *best_assigned_to*, que nos permite sempre que o *best_total_profit* é atualizado, atualizar na mesma medida qual o melhor “caminho” para obter esse novo *profit*, ou seja, a que programador a *Tarefa x* está atribuída, ou se não está atribuída.

```

285     if ((i == problem->T)) {
286         //se o meu profit atual for maior que o melhor profit, ent o melhor fica com o v
287         if (problem->total_profit > problem->best_total_profit) {
288             problem->best_total_profit = problem->total_profit;
289             problem->number_solutions = 1;
290
291             //for que percorre as tasks, e atualiza o best_assigned_to com o assigned_to
292             for (int k = 0; k < problem->T; k++)
293             {
294                 problem->task[k].best_assigned_to = problem->task[k].assigned_to;
295             }

```

Figura 7 – Comparação de variáveis, quando ramo chega ao fim

Ainda, dentro deste caso especial, temos uma expressão condicional (Fig.8), que é usada para quando a variável *I* é igual a 1, quando se corre o programa, variável essa que com esse valor tem o significado de “ignore profit”, sempre que o *total_profit* e o *best_total_profit* são iguais, incrementamos mais um ao *number_solutions*. Antes de sairmos deste bloco de código incrementamos a variável *terminal_cases* unitariamente, que servirá para contar o número de casos viáveis totais.

```

297         //usado para I = 1, ignore profit
298     } else if (problem->total_profit == problem->best_total_profit) {
299         problem->number_solutions++;
300     }
301     //incrementar o número de casos terminais
302     problem->terminal_cases++;
303     return;
304 }

```

Figura 8 – Atualizações para quando os profits são ignorados, e terminal cases

Neste momento, que as exceções já estão tratadas, podemos começar a usar as maravilhas da recursividade. Temos duas opções: avançar sem atribuir a *Tarefa*, ou tentar incluir a *Tarefa*, se conseguir avançar.

Para avançar sem atribuir a tarefa, basta chamar a própria função, no nível seguinte, *i+1* (Fig.9). Assim, estamos a não atribuir esta *Tarefa*, deste modo, vamos atualizar o *assigned_to* do nível em que nos encontramos para -1, pois como já referido anteriormente, a tarefa não será atribuída a nenhum programador.

```

306     /*avançar sem atribuir a tarefa
307     //se a tarefa nao for atribuida então assigned_to fica a -1
308     problem->task[i].assigned_to = -1;
309     recursive_function(problem, i + 1);
310

```

Figura 9 – Avançar sem atribuir a tarefa

Para tentar incluir a *Tarefa* (Fig.10), temos de executar um ciclo *for*, de zero até ao número de programadores, e ver se há algum livre, isto é, se a data até que o *Programador* está ocupado, *ending_date*, for menor que a data de começo da *Tarefa*, *starting_date*. Esta condição é deveras importante pois permite-nos de certa forma, “podar a árvore”, ou seja, permite-nos cortar certos ramos da mesma, fazendo com que o número de possibilidades totais fique mais pequeno, o que melhora de maneira considerável o tempo de execução total do algoritmo.

Caso isso se verifique a condição expressa no bloco de texto anterior, entramos no bloco de código que a condição *if* acopla, e aqui são criadas duas variáveis de cariz temporário, o *profit_tmp* e o *busy_tmp*, estas variáveis armazenam em si os valores do *total_profit* e do *busy*, respetivamente. Estas duas variáveis são importantíssimas, porque permitem repor o valor das mesmas originais às coisas elas são “filhas” quando a recursividade “volta para trás”.

```

311     /*tenta incluir a tarefa, se conseguir avança
312     for (int j = 0; j < problem->P; j++) {
313         //se houver programador livre...
314         if (problem->busy[j] < problem->task[i].starting_date) {
315             //criar variáveis tmp
316             int profit_tmp = problem->total_profit;
317             int busy_tmp = problem->busy[j];
318
319             //atualiza variáveis
320             problem->busy[j] = problem->task[i].ending_date;
321             problem->total_profit += problem->task[i].profit;
322             problem->task[i].assigned_to = j;
323
324             recursive_function(problem, i + 1);
325
326             //repor variáveis
327             problem->total_profit = profit_tmp;
328             problem->busy[j] = busy_tmp;
329             return;
330         }
331     }
332 }
```

Figura 10 - Tenta incluir a tarefa, se conseguir avança

É necessário atualizar algumas variáveis, a variável *busy* para o programador *j*, irá ficar com o valor da data final (*ending_date*) da *Tarefa i*. À variável *total_profit* é somado o *profit* da *Tarefa* em questão. E a variável *task* da *Tarefa i*, fica atribuída ao programador *j*.

Chamamos a própria função agora, no nível seguinte, para que possa prosseguir na busca do melhor *profit* pelos ramos da árvore.

Nesta zona da função voltamos a atribuir às variáveis *total_profit* e *busy* os valores que foram guardados anteriormente, e usamos um *return* para que a função possa “voltar para trás”, como já dito anteriormente.

3.3 – Algoritmos

Nesta fase, em que a função recursiva está operacional e é a sua versão final, podemos debater e analisar alguns algoritmos usados na realização da mesma. A implementação por nós feita, é uma implementação que usa a técnica *Branch & Bound*, sendo que, analisamos a possibilidade de usar outras técnicas algorítmicas, o *Divide & Conquer* e *Dynamic Programming*.

Ora, na nossa resolução inicial após fazermos um estudo do problema optamos logo por utilizar a técnica *Branch & Bound* (Fig.11), pois a nossa questão é de caráter exponencial e exige analisar bastantes possibilidades. Esta técnica algorítmica, permite-nos “retirar” possibilidades combinatórias da nossa árvore, possibilidade essas que por alguma razão são à partida inviáveis, no nosso caso, não haver nenhum programador livre na parte em que se decide incluir um programador, torna-se impossível. Deste modo conseguimos baixar o tempo de execução da nossa implementação, pois o número de possibilidades totais que iriam ser tratados fica mais pequeno, uma vez que são retiradas possibilidades inexequíveis. A figura seguinte, de forma simples consegue ilustrar o funcionamento do algoritmo.

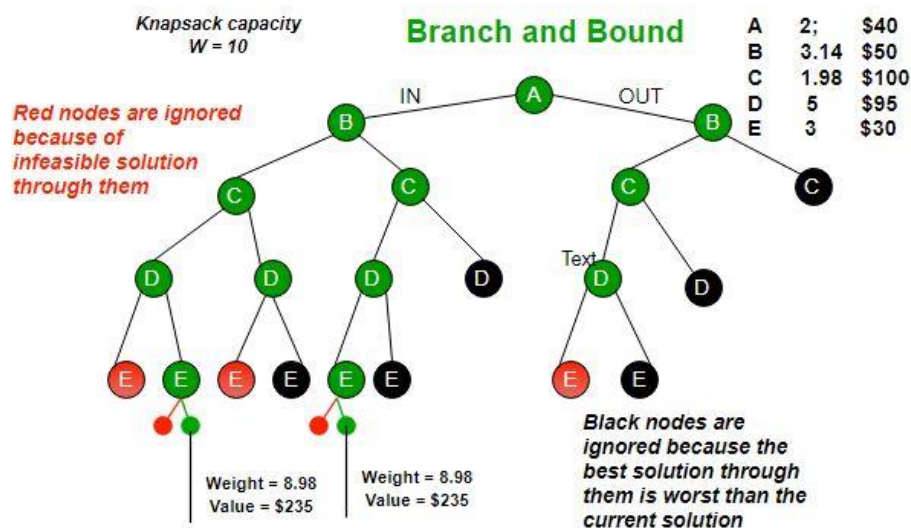


Figura 11 - Branch & Bound

Além de usar a técnica *branch and bound* para resolver o problema, abordamos duas outras técnicas algorítmicas denominadas de *divide and conquer* e *dynamic programming*.

Ora, a técnica ***Divide & Conquer*** (Fig.12) consiste em resolver um problema recursivamente através de três etapas. Assim, a primeira etapa *Divide* consiste em subdividir o problema em frações de tamanho similar. De seguida, na etapa *Conquer* fazemos chamadas recursivas até a fração do problema estar resolvida. Por fim, combinamos as soluções das frações do problema até formarmos a resposta do problema inicial, e concluímos a etapa combine. Esta implementação não nos é favorável pois as frações do nosso problema não são independentes (as *tasks* dependem umas das outras).

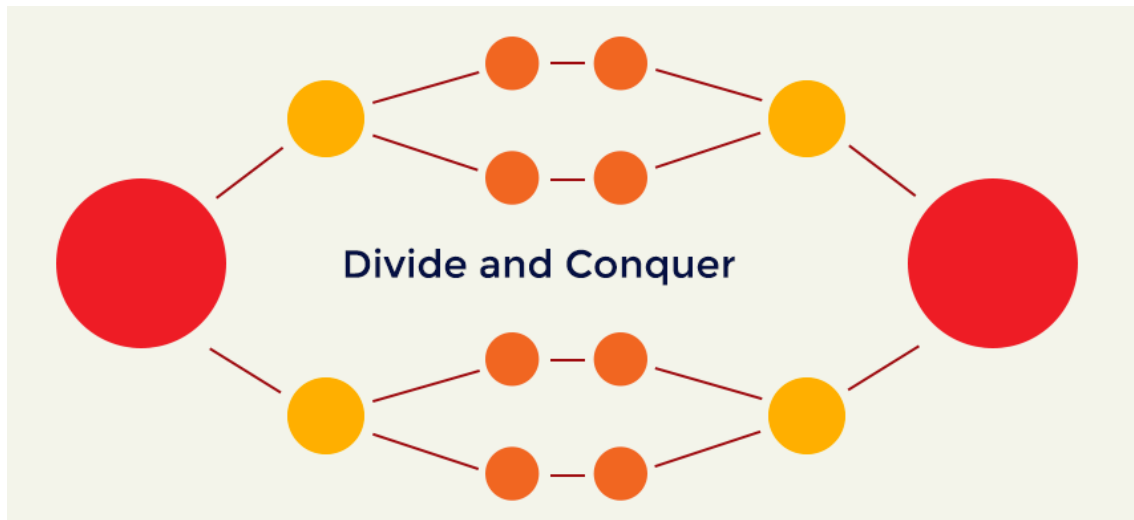


Figura 12 - Divide & Conquer (Leitura feita da esquerda para a direita)

Seguidamente, a técnica **Dynamic Programming** baseia-se em resolver as frações do problema apenas uma vez e armazenar as soluções, de modo que, quando essa informação for necessária não vai ser calculada de novo. **Esta não é a melhor implementação para resolver a nossa questão, seria boa se a nossa implementação trabalhasse só com um programador.**

3.4 – Implementação em Java do problema

Uma vez proposto numa aula prática a implementação da resolução do problema numa outra linguagem de programação, tomamos a liberdade de criar uma implementação em Java, utilizando técnicas de Orientação a Objetos, criamos 3 classes, o *Problem_t.java* para abrigar as informações relativamente ao problema, tal como na implementação em C, a classe *Task_t.java*, onde estão definidas as variáveis referentes à tarefa, e a classe *JobSelection.java*, onde está basicamente toda a implementação.

Após várias tentativas, e pesquisas, não conseguimos obter resultados válidos da implementação deste código em Java, deste modo, não conseguimos obter quaisquer resultados que seriam posteriormente comparados com a implementação em C. Achemos por bem não incluir este código no relatório pois em nada contribuiria para o mesmo, porém pode ser consultado [aqui](#).

4 - Resultados

Nesta aba do relatório serão apresentados os resultados obtidos e as conclusões dos mesmos. Relativamente ao fundamento dos resultados iremos usar gráficos, e tabelas. Para a obtenção de gráfico, implementamos código em MatLab, e para a obtenção de tabelas, com a implementação de código Java.

Após o `job_selection_do_all.bash` ter terminado, executamos o script `extract_data.bash` (Fig. 13), disponibilizado na página online da unidade curricular. Este script, irá criar dois ficheiros `.txt`, cada um com 3 colunas de informação, um deles com as *Tarefas, Programadores, e profit*, e outro com *Tarefas, Programadores, e Tempos de Execução*. Este script foi executado 3 vezes, uma por cada elemento do grupo com o respetivo número mecanográfico. É necessário antes de executar o script atribuir permissões.

```

1  #!/bin/bash
2  # script bash onde guardamos em 2 ficheiros .txt os tempos de execução, e os profits de cada task
3
4  cd ../098491/
5
6  grep "Solution time =" *.txt | sed -e 's/_0.txt:Solution time =/' -e 's/_/ /' >../Resultados/temposExecucao98491.txt
7  grep "Best Profit =" *.txt | sed -e 's/_0.txt:Best Profit =/' -e 's/_/ /' >../Resultados/totalsProfit98491.txt
8

```

Figura 13 – `extract_data.bash`

4.1 - Tabelas

No que toca à criação das tabelas, decidimos fazer uma [implementação em Java](#), implementação essa que abre os ficheiros criados pelo script `extract_data.bash` em modo leitura, e cria novos ficheiros com a informação devidamente organizada e visualmente agradável.

Como conseguimos resolver o problema até 40 *Tarefas e 8 Programadores*, temos um total de 292 tarefas feitas, o que implica que a tabela seja muito comprida, impossibilitando assim que a mesma fique completa neste relatório. As tabelas completas podem ser consultadas [aqui](#).

A tabela seguinte (Fig. 14), é representativa do elemento do grupo com o número mecanográfico 98491.

Profit - 98491		
Tarefas	Programadores	Profit
01	01	1187
02	01	4588
02	02	2590
03	01	2634
03	02	4770
03	03	2481
04	01	3573
04	02	5899
04	03	5213
04	04	9826
05	01	5882
05	02	5972
05	03	6436
05	04	11044
05	05	12666
06	01	8521
06	02	8812
06	03	8409

Figura 14 – Tabela dos profits para o número mecanográfico 98491

A tabela seguinte (Fig. 15), é representativa do elemento do grupo com o número mecanográfico 98495.

Profit - 98495		
Tarefas	Programadores	Profit
01	01	513
02	01	4496
02	02	1571
03	01	2823
03	02	2726
03	03	8865
04	01	5475
04	02	6287
04	03	8294
04	04	7339
05	01	8740
05	02	8305
05	03	10236
05	04	8382
05	05	10621
06	01	8157
06	02	7545
06	03	9471

Figura 15 - Tabela dos profits para o número mecanográfico 98495

A seguinte tabela (Fig. 16), representa os *profits* obtidos pelo elemento do grupo com o número mecanográfico 98629.

Profit - 98629		
Tarefas	Programadores	Profit
01	01	1004
02	01	3548
02	02	4370
03	01	2772
03	02	5163
03	03	5270
04	01	6679
04	02	5906
04	03	6346
04	04	4504
05	01	5459
05	02	4812
05	03	16902
05	04	6933
05	05	15023
06	01	4636
06	02	5848
06	03	10081

Figura 16 - Tabela dos profits para o número mecanográfico 98629

4.2 – Gráficos

Em relação aos gráficos criados, criamos gráficos com e para diversos propósitos, para demonstrar o profit que cada elemento do grupo alcançou, para ilustrar os tempos de execução por parte de cada elemento, e ainda gráficos para a parte de se ignorar o profit. Todos os gráficos foram criados com recurso ao MatLab, ferramenta bastante útil, pois permite de forma simples criar estas figuras, e assim tirar conclusões de uma forma muito mais simples e rápida. Todo o código implementado pode ser consultado [aqui](#), ou então no ponto 5 do relatório – [Apêndice](#). As imagens dos gráficos podem também ser consultadas [aqui](#).

4.2.1 – Profit

Nas três figuras seguintes (Fig. 17, 18, 19) estão representados os **gráficos com os resultados dos profits**, para cada número mecanográfico, há 2 gráficos, um que mostra em 2 eixos os profits e as tarefas, e outro em 3 eixos, com o profit, o número de tarefas a variar entre 1 e 40 e o número de programadores a variar entre 1 e 8.

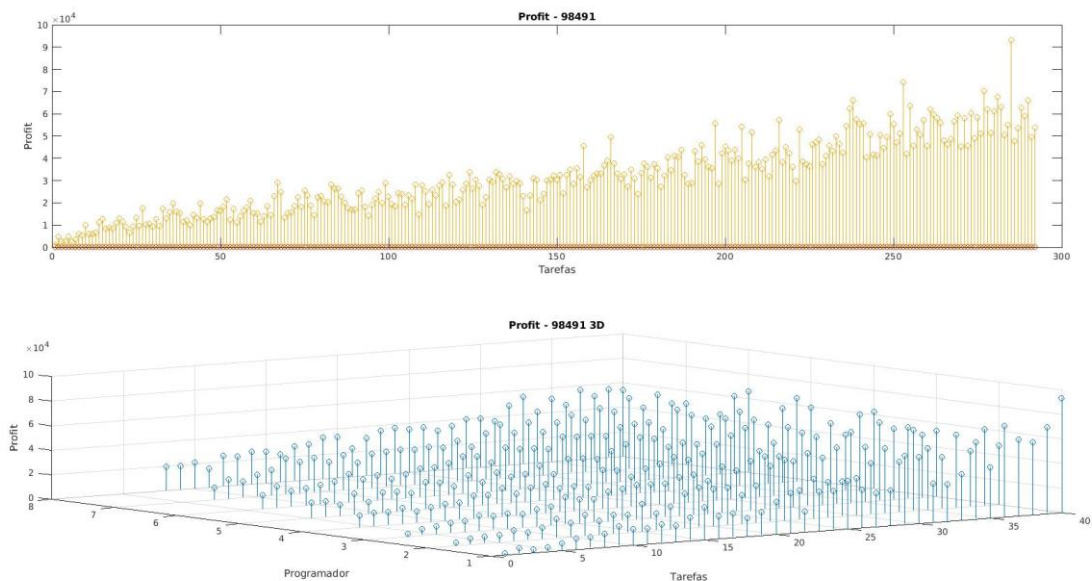


Figura 17 – Gráficos em 2D e 3D dos profits do número mecanográfico 98491

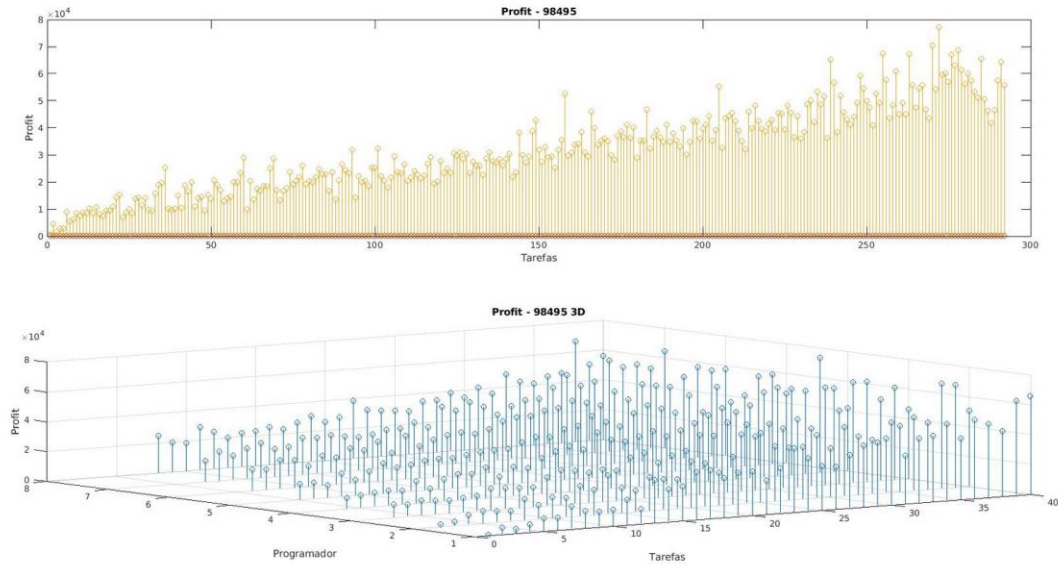


Figura 18 - Gráficos em 2D e 3D dos profits do número mecanográfico 98495

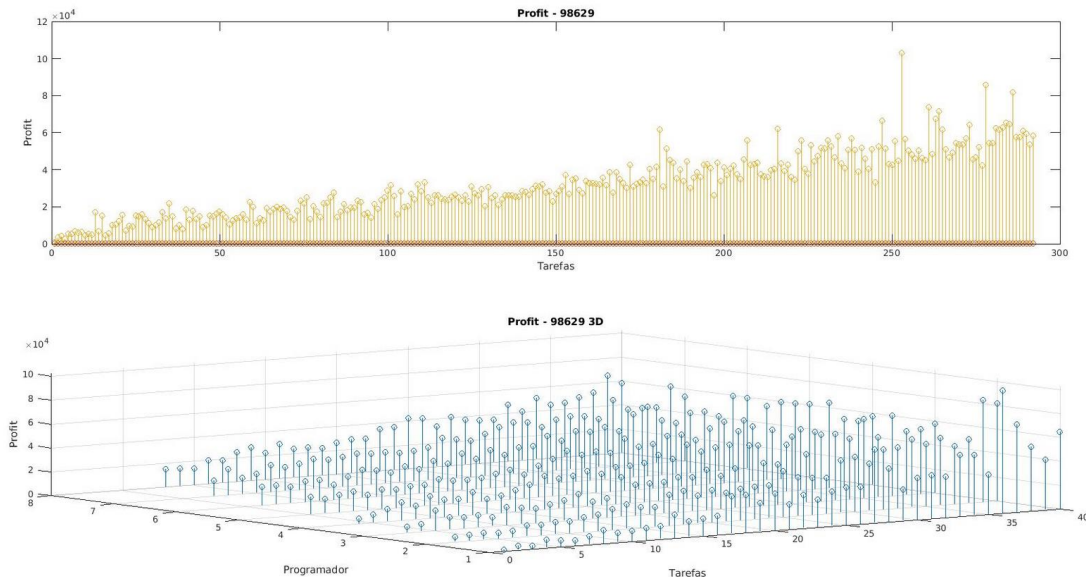


Figura 19 – Gráficos em 2D e 3D dos profits do número mecanográfico 98629

Relativamente a estes gráficos conseguimos concluir que os profits crescem à medida que o número de tarefas aumenta, é um crescimento não linearmente perfeito, mas pela observação direta do gráfico com 2 eixos, nota-se facilmente esse crescimento, ao longo das tarefas. Importante também denotar que por norma, quanto maior o número de tarefas e de programadores maior é o profit conseguido, no entanto como o gráfico mostra, para algumas tasks, o profit foge à regra, sendo bastante mais alto ou baixo, em relação ao que deveria ser se seguisse o “padrão” anteriormente falado. A aplicação de um gráfico em 3 dimensões permite, observar o crescimento do profit também em relação ao número de programadores, conjugado com o número de tarefas.

Aproveitamos ainda para realizar fazer a regressão linear, para cada um dos elementos do grupo (Fig.20), o R^2 de cada elemento está expresso na legenda do gráfico. Não apresentam um R^2 muito alto, porém tendo em conta que os profits advêm de um processo de cálculo de operações aleatórias, e que dependem em grande parte do número de programadores, e de tarefas, podemos concluir que são valores bastante aceitáveis, e tal como as retas mostram na figura, crescem à medida que o número de tarefas aumenta.

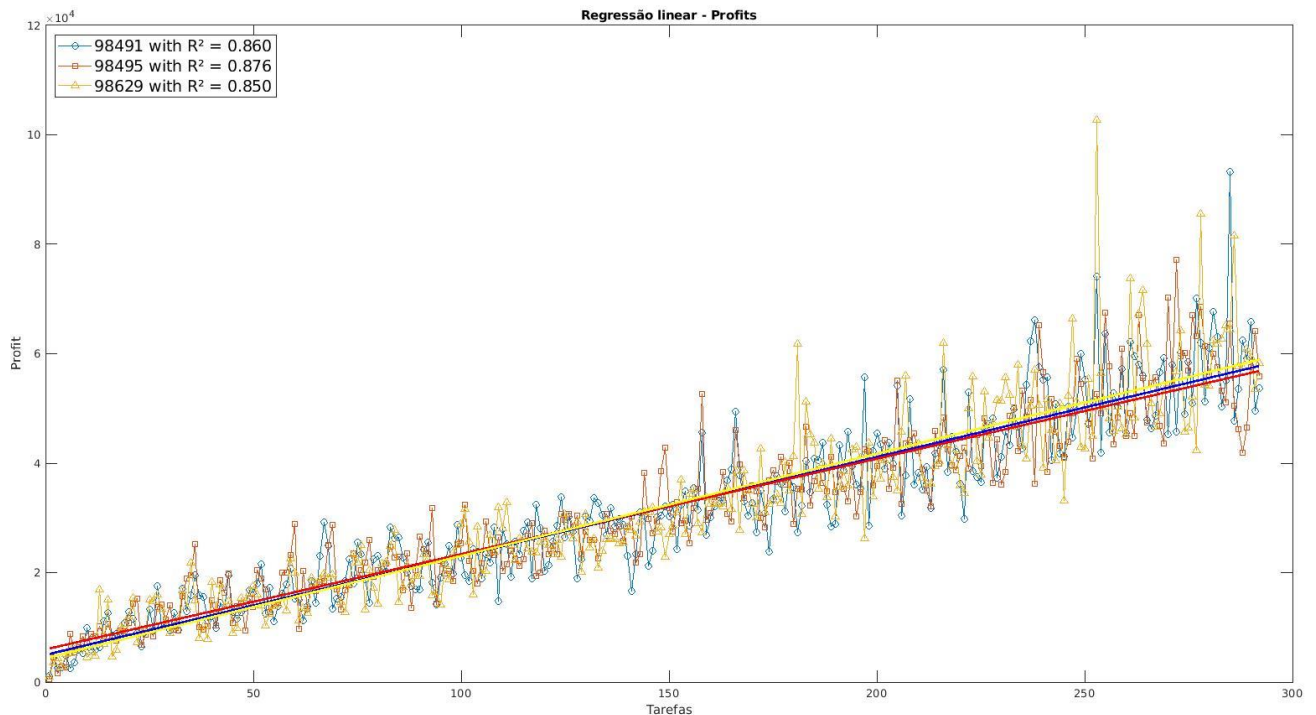


Figura 20 – Regressão linear

4.2.2 – Tempos de execução

Relativamente à matéria de tempos de execução, criamos também 2 gráficos um com 2 eixos (Fig.21), onde é mostrada o crescimento do tempo de execução, em escala logarítmica, à medida que o número de tasks aumenta, e também um gráfico com 3 dimensões (Fig.22) onde é visível o crescimento dos tempos de execução, em escala logarítmica, em relação ao número de programadores e de tarefas. Ambos os gráficos são representativos dos 3 elementos do grupo.

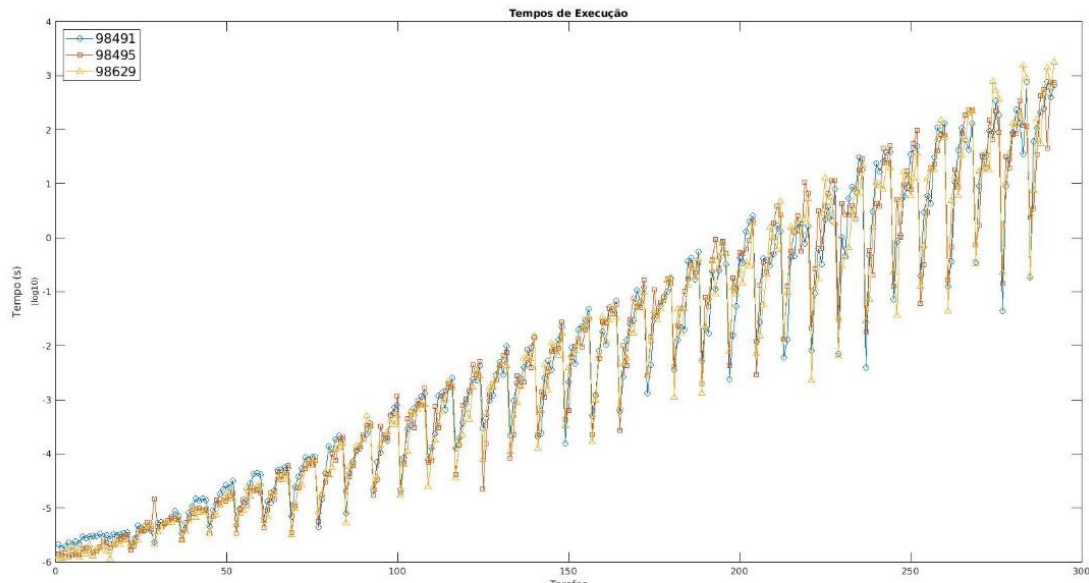


Figura 21 - Gráfico de tempos de execução

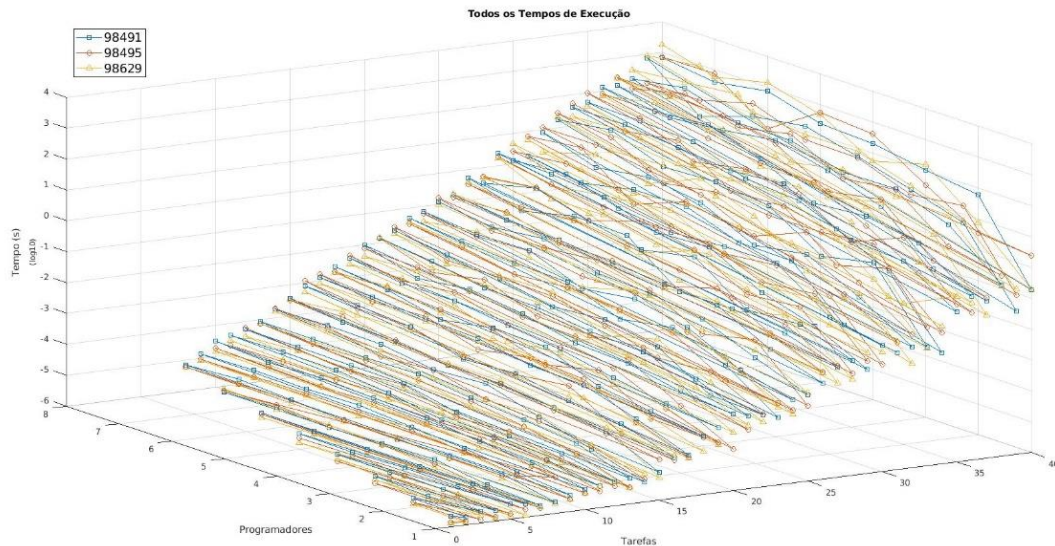


Figura 22 - Gráfico 3D dos tempos de execução

Quanto aos tempos de execução, podemos tirar algumas conclusões, das quais destacamos: Quanto maior for o número de Tarefas, maior é a discrepância de tempos de execução entre quando há um programador e quando há 8, isto acontece, pois, o número de possibilidades com o aumento do número de programadores aumenta. Podemos concluir também que o tempo de execução cresce de forma exponencial como a figura 21 mostra, pois o tempo encontra-se na escala logarítmica como a figura mostra. Relativamente ao gráfico 3D, onde estão expressos todos os tempos de execução, em relação ao número de programadores e de tarefas a serem feitas, observamos de forma evidente o crescimento dos tempos à medida que o número de tarefas aumenta e à medida que o número de programadores aumenta. Podemos ainda concluir, e isto até mais por observação ao executar o código, que

até um número de tarefas +/- igual a 34, a resolução do programa é bastante rápida, sendo que para números adiante demora um pouco mais de tempo.

4.2.3 – Ignorar Profits

Decidimos por bem, correr a nossa implementação para quando os profits são ignorados, visto que durante a criação da função recursiva, fizemos código para tratar esta eventualidade, criamos os seguintes gráficos (Fig. 23, 24, 25). Para extrair a informação, criamos o *script bash extract_without_profit.sh*, que podem ser consultados [aqui](#), onde retiramos desta forma as *Programing Taks* e os *Terminal Cases*.

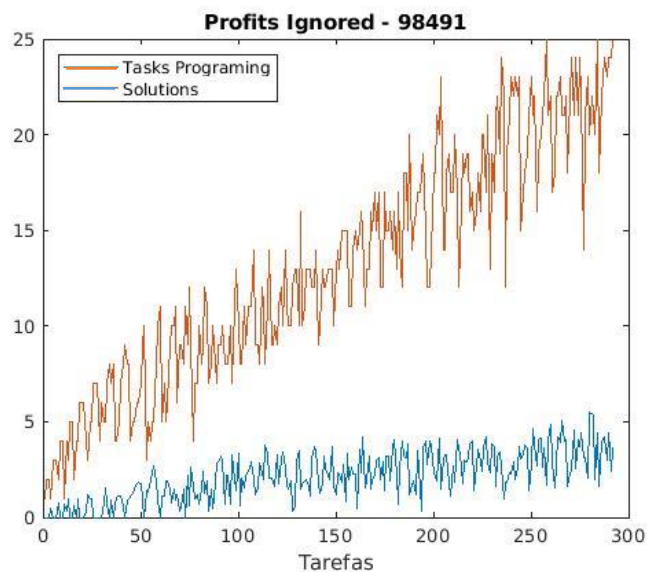


Figura 23 - Profits ignorados - 98491

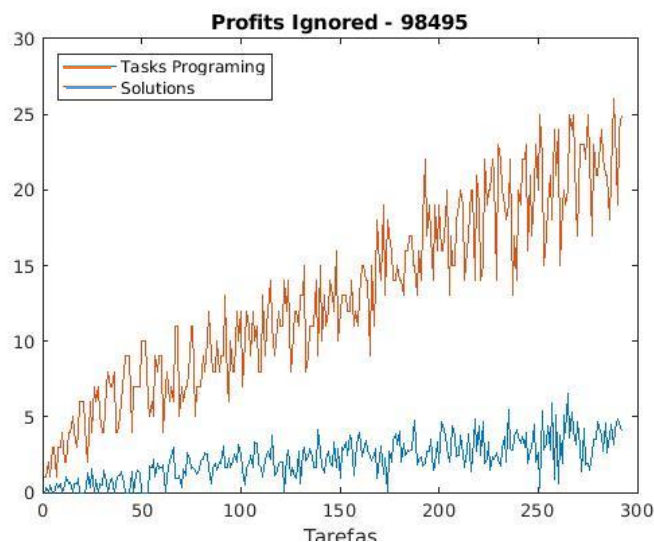


Figura 24 - Profits ignorados - 98495

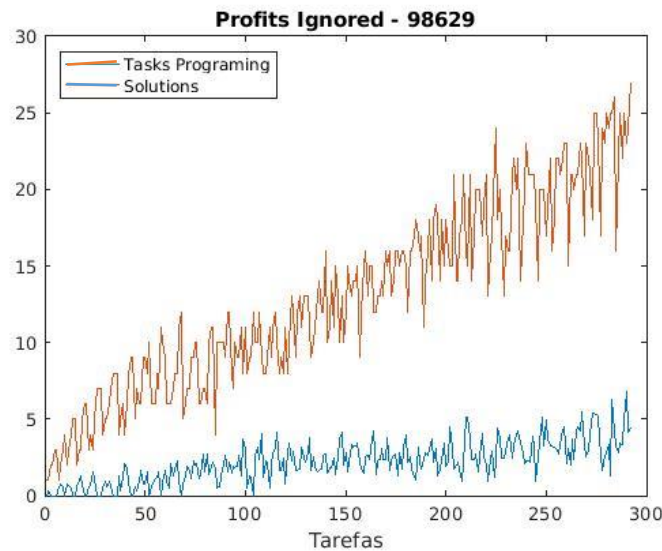


Figura 25 - Profits ignorados - 98629

Em relação aos gráficos com os profits ignorados, temos 2 variantes a analisar, as Tasks Programing e as Solutions. Com uma observação cuidada dos gráficos, consegue-se concluir que relativamente as Tasks Programing (linha a azul, e em escala logarítmica, para que se consiga observar as 2 variantes ao mesmo tempo de forma focada) o seu crescimento ao longo no número de tarefas é pequeno, é um crescimento bastante lento ao longo de toda a resolução do problema, as oscilações que se observam são referentes ao número de programadores, valores mais baixos para um número de programadores mais baixo, e valores mais altos para números de programadores mais altos, isto para todos os números mecanográficos.

Quanto às Solutions, esta variante cresce um pouco mais rápido, as oscilações são provocadas pelo número de programadores, o seu crescimento segue um padrão, quanto maior o número de tarefas mais Solutions haverá tal como os gráficos mostram, estas conclusões são válidas também para os 3 gráficos apresentados.

4.2.4 – Histogramas sobre o Profit

Relativamente a este ponto, o essencial é mostrar os histogramas criados para cada número mecanográfico (Fig.26, 27, 28), concluir através deles se representam uma distribuição normal, e contar o número total de tasks.

Como já dito anteriormente, todos os membros executaram a solução até 40 tarefas e 8 programadores, sendo desta forma, o total de tasks igual a 292 para todos.

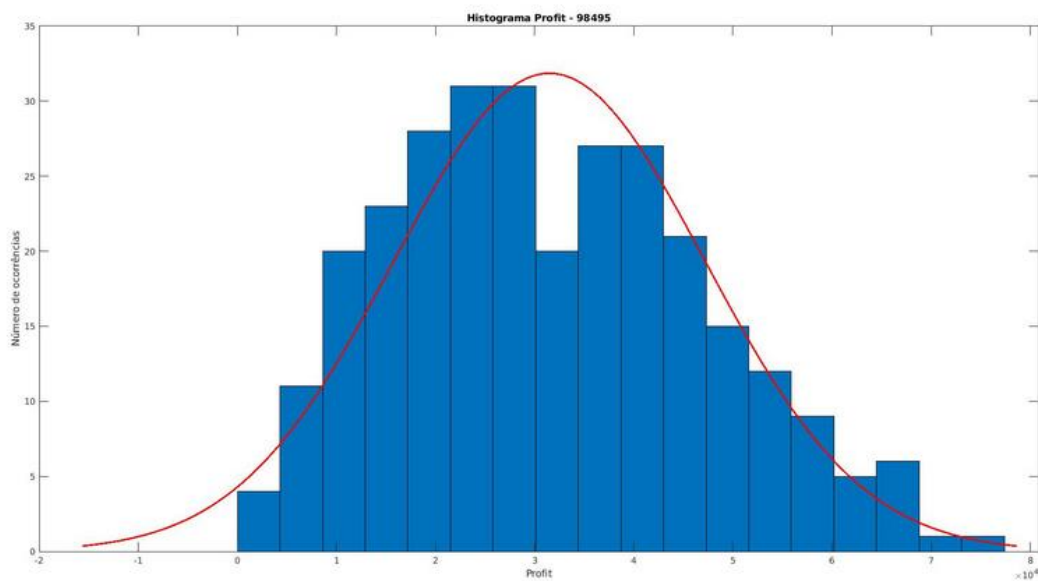


Figura 26 – Histograma com o número de ocorrências de cada profit para o número mecanográfico 98495

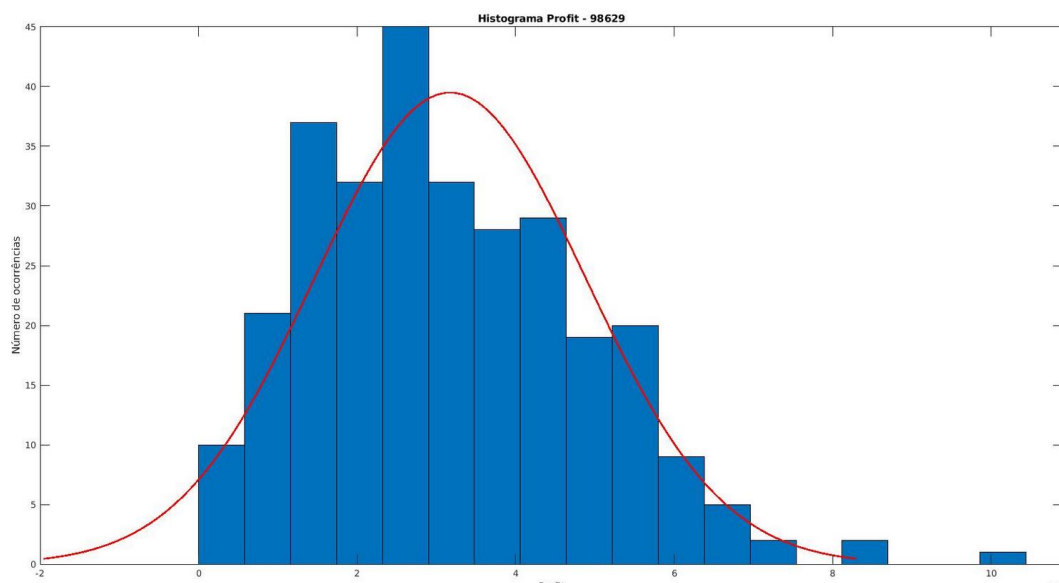


Figura 27 - Histograma com o número de ocorrências de cada profit para o número mecanográfico 98629

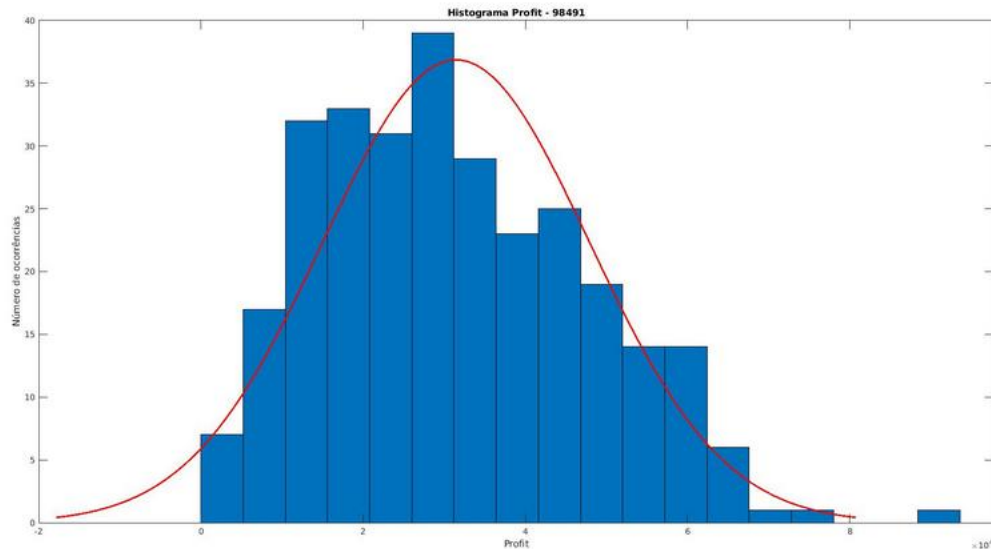


Figura 28 - Histograma com o número de ocorrências de cada profit para o número mecanográfico 98491

No que diz respeito aos histogramas apresentados, podemos concluir que uma grande maioria dos profits se encontram numa zona mais central dos mesmos, no entanto, através da análise dos histogramas consegue-se concluir que os mesmos não apresentam uma distribuição normal. A linha a vermelho nos gráficos ajuda melhor a chegar a estas conclusões, para que fosse normal, o histograma teria de respeitar a “forma” que a linha vermelha espelha, como podemos ver isso não acontece, logo a distribuição dos profits não poderá ser considerada normal.

5-Apêndice

5.1 -Tabelas

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 public class Tabelas {
7
8     public static void main(String[] args) {
9
10         String[] fileNameInput = { "./Resultados/totalsProfit98491.txt", "./Resultados/totalsProfit98495.txt", "./Resultados/totalsProfit98629.txt", };
11         String[] fileNameOutput = { "./Resultados/Tabelas/Profit98491.txt", "./Resultados/Tabelas/Profit98495.txt", "./Resultados/Tabelas/Profit98629.txt", };
12         String[] numMec = {"98491", "98495", "98629"};
13
14         for (int i = 0; i < fileNameInput.length; i++) {
15
16             try (Scanner input = new Scanner(new File(fileNameInput[i]))) {
17                 try (PrintWriter out = new PrintWriter(new File(fileNameOutput[i]))) {
18                     out.printf("+-----+\n");
19                     out.printf("                Profit - %s                |\n", numMec[i]);
20                     out.printf("+-----+\n");
21                     out.printf(" %10s | %16s | %10s |\n", "Tarefas", "Programadores", "Profit");
22                     out.printf("+-----+\n");
23
24                     while (input.hasNextLine()) {
25                         String linha = input.nextLine();
26                         String[] tab = linha.split(" ");
27
28                         out.printf("+-----+\n");
29                         out.printf(" | %8s | %12s | %10s |\n", tab[0], tab[1], tab[2]);
30                     }
31                     out.printf("+-----+\n");
32
33                 } catch (FileNotFoundException e) {
34                     System.err.println(e.getMessage());
35                 }
36             } catch (FileNotFoundException e) {
37                 System.err.println(e.getMessage());
38             }
39         }
40     }
41 }
42
43
44

```

Figura 29 – Código Java que gera as tabelas com os profits

5.2 - Matlab

```
1 clear all; close all;
2
3
4 file98491 = load("../temposExecucao98491.txt");
5 file98495 = load("../temposExecucao98495.txt");
6 file98629 = load("../temposExecucao98629.txt");
7
8 x = 1:length(file98491(:,1));
9
10 figure(1);
11 plot(x, log10(file98491(:,3)), '-o');
12 hold on;
13 plot(x, log10(file98495(:,3)), '-s');
14 plot(x, log10(file98629(:,3)), '-^');
15 hold off
16 title 'Tempos de Execução'
17 xlabel 'Tarefas'
18 ylabel ({'Tempo (s)', '_{log10}'});
19 legend ({'98491', '98495', '98629'}, 'Location', 'northwest', 'FontSize', 14);
```

Figura 30 – Código MatLab gerador do gráfico com os tempos de execução


```

1 clear all; close all;
2
3 % Gráfico com os tempos de execução
4
5 file98491 = load("../temposExecucao98491.txt");
6 file98495 = load("../temposExecucao98495.txt");
7 file98629 = load("../temposExecucao98629.txt");
8
9 T98491 = file98491(:,1);
10 P98491 = file98491(:,2);
11 t98491 = log10(file98491(:,3));
12
13 T98495 = file98495(:,1);
14 P98495 = file98495(:,2);
15 t98495 = log10(file98495(:,3));
16
17 T98629 = file98629(:,1);
18 P98629 = file98629(:,2);
19 t98629 = log10(file98629(:,3));
20
21 figure(1)
22 plot3(T98491, P98491, t98491, '-s');
23 hold on;
24 plot3(T98495, P98495, t98495, '-o');
25 plot3(T98629, P98629, t98629, '-^');
26 hold off;
27
28 title 'Todos os Tempos de Execução'
29 xlabel ('Tarefas')
30 ylabel ('Programadores')
31 zlabel ({'Tempo (s)', ' {(log10)}'});
32 legend ({'98491', '98495', '98629'}, 'Location', 'northwest', 'FontSize', 14);
33 grid on;

```

Figura 31 - Código MatLab, tempos execução 3D

```

1 clear all;
2 close all;
3
4 file98491 = load("../programing98491.txt");
5 file98491_2 = load("../terminal98491.txt");
6
7 file98495 = load("../programing98495.txt");
8 file98495_2 = load("../terminal98495.txt");
9
10 file98629 = load("../programing98629.txt");
11 file98629_2 = load("../terminal98629.txt");
12
13 eixox = 1:292;
14
15 figure(1)
16 y98491 = log10(file98491_2(:, 3));
17 x98491 = file98491(:,3);
18 plot(eixox, y98491);
19 hold on;
20 plot(eixox, x98491);
21 title 'Profits Ignored - 98491'
22 legend('Tasks Programing','Solutions', 'Location', 'northwest');
23
24 figure(2)
25 y98495 = log10(file98495_2(:, 3));
26 x98495 = file98495(:,3);
27 plot(eixox, y98495);
28 hold on;
29 plot(eixox, x98495);
30 title 'Profits Ignored - 98495'
31 legend('Tasks Programing','Solutions', 'Location', 'northwest');
32
33 figure(3)
34 y98629 = log10(file98629_2(:, 3));
35 x98629 = file98629(:,3);
36 plot(eixox, y98629);
37 hold on;
38 plot(eixox, x98629);
39 title 'Profits Ignored - 98629'
40 legend('Tasks Programing','Solutions', 'Location', 'northwest');

```

Figura 32 - Código MatLab, Ignore Profit

```

1 clear all; close all;
2
3 % Gráfico com todos os profits profits, e faz a regressão linear
4 file98491 = load("../totalsProfit98491.txt");
5 file98495 = load("../totalsProfit98495.txt");
6 file98629 = load("../totalsProfit98629.txt");
7
8 profit98491 = file98491(:,3);
9 profit98495 = file98495(:,3);
10 profit98629 = file98629(:,3);
11
12 x = 1:length(profit98491);
13
14 pp98491 = polyfit(x,profit98491,1);
15 pp98495 = polyfit(x,profit98495,1);
16 pp98629 = polyfit(x,profit98629,1);
17
18 plot(x, profit98491, '-o');
19 hold on;
20 plot(x, profit98495, '-s');
21 plot(x, profit98629, '-^');
22
23 plot(x, polyval(pp98491, x),'-b', "LineWidth", 2);
24 plot(x, polyval(pp98495, x),'-r', "LineWidth", 2);
25 plot(x, polyval(pp98629, x),'-y', "LineWidth", 2);
26 hold off;
27
28 xlabel 'Tarefas'
29 ylabel 'Profit'
30 title 'Regressão linear - Profits'
31 legend({'98491', '98495', '98629'},'Location', 'northwest', 'FontSize', 14);

```

Figura 33 – Código MatLab usado para criar o gráfico com a regressão linear dos profits

```

1 clear all;
2 close all;
3
4 file98491 = load("../totalsProfit98491.txt");
5 file98495 = load("../totalsProfit98495.txt");
6 file98629 = load("../totalsProfit98629.txt");
7
8
9 figure(1)
10 profit98491 = file98491(:, 3);
11 hist(profit98491);
12 title 'Histograma Profit - 98491'
13 xlabel 'Profit'
14 ylabel 'Número de ocorrências'
15
16 figure(2)
17 profit98495 = file98495(:, 3);
18 hist(profit98495);
19 title 'Histograma Profit -98495'
20 xlabel 'Profit'
21 ylabel 'Número de ocorrências'
22
23 figure(3)
24 profit98629 = file98629(:, 3);
25 hist(profit98629);
26 title 'Histograma Profit - 98629'
27 xlabel 'Profit'
28 ylabel 'Número de ocorrências'

```

Figura 34 – Código gerador dos histogramas com o número de ocorrências

```

1  clear all;close all;
2
3  % Profit - 98491
4  file98491 = load("../totalsProfit98491.txt");
5
6  figure(1)
7  subplot(2,1,1);
8  stem(file98491);
9  title 'Profit - 98491'
10 xlabel 'Tarefas'
11 ylabel 'Profit'
12 hold on;
13
14 X = file98491(:, 1);
15 Y = file98491(:, 2);
16 Z = file98491(:, 3);
17
18 subplot(2,1,2);
19 stem3(X, Y, Z);
20 title 'Profit - 98491 3D'
21 xlabel 'Tarefas'
22 ylabel 'Programador'
23 zlabel 'Profit'
24 hold off;
25
26 % Profit - 98495
27 file98495 = load("../totalsProfit98495.txt");
28
29 figure(2)
30 subplot(2,1,1);
31 stem(file98495);
32 title 'Profit - 98495'
33 xlabel 'Tarefas'
34 ylabel 'Profit'
35
36 hold on;
37 X = file98495(:, 1);
38 Y = file98495(:, 2);
39 Z = file98495(:, 3);
40
41 subplot(2,1,2);
42 stem3(X, Y, Z);
43 title 'Profit - 98495 3D'
44 xlabel 'Tarefas'
45 ylabel 'Programador'
46 zlabel 'Profit'
47 hold off;
48
49
50 % Profit - 98629
51 file98629 = load("../totalsProfit98629.txt");
52 figure(3)
53
54 subplot(2,1,1);
55 stem(file98629);
56 title 'Profit - 98629'
57 xlabel 'Tarefas'
58 ylabel 'Profit'
59 hold on;
60
61 X = file98629(:, 1);
62 Y = file98629(:, 2);
63 Z = file98629(:, 3);
64
65 subplot(2,1,2);
66 stem3(X, Y, Z);
67 title 'Profit - 98629 3D'
68 xlabel 'Tarefas'
69 ylabel 'Programador'
70 zlabel 'Profit'

```

Figura 35 – Criação dos gráficos 3d com os profits

5.3- Ficheiros Bash

```

1  #!/bin/bash
2  # script bash onde guardamos em 2 ficheiros .txt os tempos de execução, e os profits de cada task
3
4  cd ../098491/
5
6  grep "Solution time =" *.txt | sed -e 's/_0.txt:Solution time =//' -e 's/_/ /' >../Resultados/temposExecucao98491.txt
7  grep "Best Profit =" *.txt | sed -e 's/_0.txt:Best Profit =//' -e 's/_/ /' >../Resultados/totalsProfit98491.txt
8
9
10

```

Figura 36 – Código bash, *extract_data_bash*

```

1  #!/bin/bash
2  # script bash onde guardamos em 2 ficheiros .txt os tempos de execução, e os profits de cada task
3
4  cd ../098629_1/
5
6  grep "Programing Tasks =" *.txt | sed -e 's/_1.txt:Programing Tasks =//' -e 's/_/ /' >../Resultados/programing98629.txt
7  grep "Terminal Cases =" *.txt | sed -e 's/_1.txt:Terminal Cases =//' -e 's/_/ /' >../Resultados/terminal98629.txt
8
9
10

```

Figura 37 – Código bash, *extract_without_profit.sh*

5.4 – Criação de novas variáveis

Foram criadas as variáveis, `best_assigned_to`, `best_total_profit`, `terminal_cases` e `number_solutions`.

```

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)
    int best_assigned_to; // Fazer o best_assigned_to com o caminho que tem o melhor profit (for em task_t)
} 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
    int best_total_profit; // best 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
    unsigned long int terminal_cases; // number of analized cases
    unsigned long int number_solutions; // number of solutions, used when I is 1
} problem_t;

```

Figura 38 - Novas variáveis

5.5 – Função recursiva

```

355 //por motivos de tempo de execucao, para não estar sempre a fazer a comparação I == 0, o que é desnecessário
356 //inicializamos as seguintes variáveis aqui, fora da chamada da função, neste bloco
357 {
358     problem->total_profit = 0;
359     problem->best_total_profit = 0;
360     problem->terminal_cases = 0;
361     //Inicializar busy a '-1'
362     for (int k = 0; k < problem->P; k++) {
363         problem->busy[k] = -1;
364     }
365 }
366
367 recursive_function(problem, 0);

```

Figura 39 – Inicialização de variáveis fora da função recursiva

```

void recursive_function(problem_t *problem, int i) {

    if ((i == problem->T)) {
        //se o meu profit atual for maior que o melhor profit, ent o melhor fica com o valor do atual
        if (problem->total_profit > problem->best_total_profit) {
            problem->best_total_profit = problem->total_profit;
            problem->number_solutions = 1;

            //for que percorre as tasks, e atualiza o best_assigned_to com o assigned_to cada vez que há um novo melhor profit
            for (int k = 0; k < problem->T; k++)
            {
                problem->task[k].best_assigned_to = problem->task[k].assigned_to;
            }

            //usado para I = 1, ignore profit
        } else if (problem->total_profit == problem->best_total_profit) {
            problem->number_solutions++;
        }
        //incrementar o número de casos terminais
        problem->terminal_cases++;
        return;
    }

    //avançar sem atribuir a tarefa
    //se a tarefa nao for atribuida então assigned_to fica a -1
    problem->task[i].assigned_to = -1;
    recursive_function(problem, i + 1);

    //tenta incluir a tarefa, se conseguir avança
    for (int j = 0; j < problem->P; j++) {
        //se houver programador livre...
        if (problem->busy[j] < problem->task[i].starting_date) {
            //criar variáveis tmp
            int profit_tmp = problem->total_profit;
            int busy_tmp = problem->busy[j];

            //atualiza variaveis
            problem->busy[j] = problem->task[i].ending_date;
            problem->total_profit += problem->task[i].profit;
            problem->task[i].assigned_to = j;

            recursive_function(problem, i + 1);

            //repoe variaveis
            problem->total_profit = profit_tmp;
            problem->busy[j] = busy_tmp;
            return;
        }
    }
}

```

Figura 40 - Função recursiva

6 - Conclusão

Nesta fase final do relatório podemos dizer de uma forma bastante segura que conseguimos implementar uma solução suficientemente notável, pois conseguimos de forma convicta resolver o problema em questão e ainda incrementar algumas ferramentas que nos ajudassem a dar mais informação sobre os profits (quer para não era ignorado, quer para quando era).

Em relação à solução encontrada, esta foi uma solução que nos levou a resultados bastante satisfatórios, conciliando recursividade, a estratégia algorítmica Branch & Bound, e pesquisa exaustiva, conseguimos implementar uma função no mínimo eficiente e com um elevado grau de competência computacional, que nos retorna a melhor maneira, ou seja, a maneira para obter lucro, sem sobreposição de programadores.

Dos objetivos propostos na introdução, estamos em condições de afirmar que todos foram alcançados com bastante sucesso, menos o objetivo da implementação da solução em Java, foi feita uma implementação, porém não conseguimos obter resultados válidos nem no mínimo satisfatórios.

Sentimos alguma dificuldade na elaboração do código da função recursiva pela nossa falta de experiência a trabalhar com a linguagem C, porém essa dificuldade foi diminuindo à medida que tínhamos mais aulas, e que trabalhávamos mais na matéria.

7 - Bibliografia

Silva, Tomás Oliveira e. Lecture notes: Algorithms and Data Structure (AED - Algoritmos e Estruturas de Dados), LEI, MIEC, 2020/2021

<https://www.geeksforgeeks.org/branch-and-bound-algorithm/> [3/12/2020]

<https://www.geeksforgeeks.org/divide-and-conquer/> [3/12/2020]

<https://www.geeksforgeeks.org/dynamic-programming/> [3/12/20]

<https://pt.stackoverflow.com/> [5/12/2020]

<https://www.mathworks.com/help/matlab/> [22/12/2020]