

**Algoritmos e Estruturas de Dados**

**Job Selection**

**Professores:**

Tomás Oliveira e Silva ([tos@ua.pt](mailto:tos@ua.pt))  
Pedro Lavrador ([plavrador@ua.pt](mailto:plavrador@ua.pt))

**Pedro** **Sobral**, 98491 – XX%

**André Freixo**, 98495 – XX%

**Marta Fradique**, 98626 – XX%

XX/01/2021

Índice

[**1 – Introdução** 3](#_Toc60593998)

[**2 - Introdução ao Problema** 4](#_Toc60593999)

[2.1 - Resumo 4](#_Toc60594000)

[2.2 – Compilação e Execução 4](#_Toc60594001)

[**3 – Resolução do Problema** 6](#_Toc60594002)

[3.1 - Inicialização 6](#_Toc60594003)

[3.2 – Função Recursiva 6](#_Toc60594004)

[3.3 – Algoritmos 10](#_Toc60594005)

[3.4 – Implementação em Java do problema 11](#_Toc60594006)

[**4 - Resultados** 12](#_Toc60594007)

[4.1 - Tabelas 12](#_Toc60594008)

[4.2 – Gráficos 16](#_Toc60594009)

[4.2.1 – Profit 16](#_Toc60594010)

[4.2.2 – Histogramas Profit 19](#_Toc60594011)

[4.2.3 – Ignorar Profits 21](#_Toc60594012)

[4.2.4 – Tempos de Execução 23](#_Toc60594013)

[**5 - Apêndice** 26](#_Toc60594014)

[5.1 - Tabelas 26](#_Toc60594015)

[5.2 - Matlab 26](#_Toc60594016)

[5.3 – Ficheiro BASH 34](#_Toc60594017)

[**6 - Conclusão** 35](#_Toc60594018)

[**7 - Bibliografia** 36](#_Toc60594019)

# **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](https://github.com/TheScorpoi/AED_Trabalho01)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 arvore 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 saí da árvore, reduzindo desta forma o número total de possibilidades e aumentando em larga escala o tempo de execução. Á medida que a árvore vai sendo criada e podada, o *profit* vai sendo comparado com o melhor atual, entro 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*.

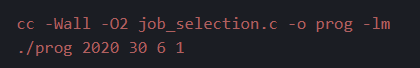
Posto isto, para compilar basta:

*cc -Wall -O2 job\_selection.c -o prog*

Para correr a compilação basta:

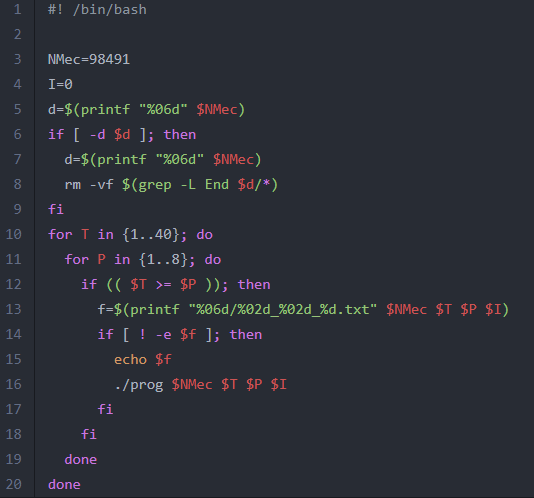
*./prog NMec T P I*

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.



Figura

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.*

**

Figura

**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).

**

Figura

# **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

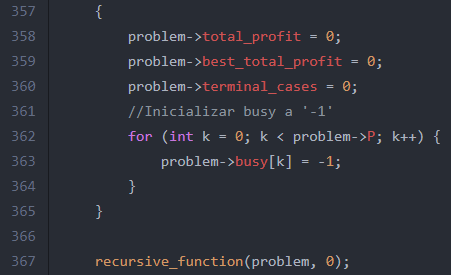
A nossa função recursiva é dotada de dois argumentos à cabeça, um deles o problema que a mesma irá tratar, o outro onde irá começar.



Figura

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, 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.

Figura



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.XX), 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.

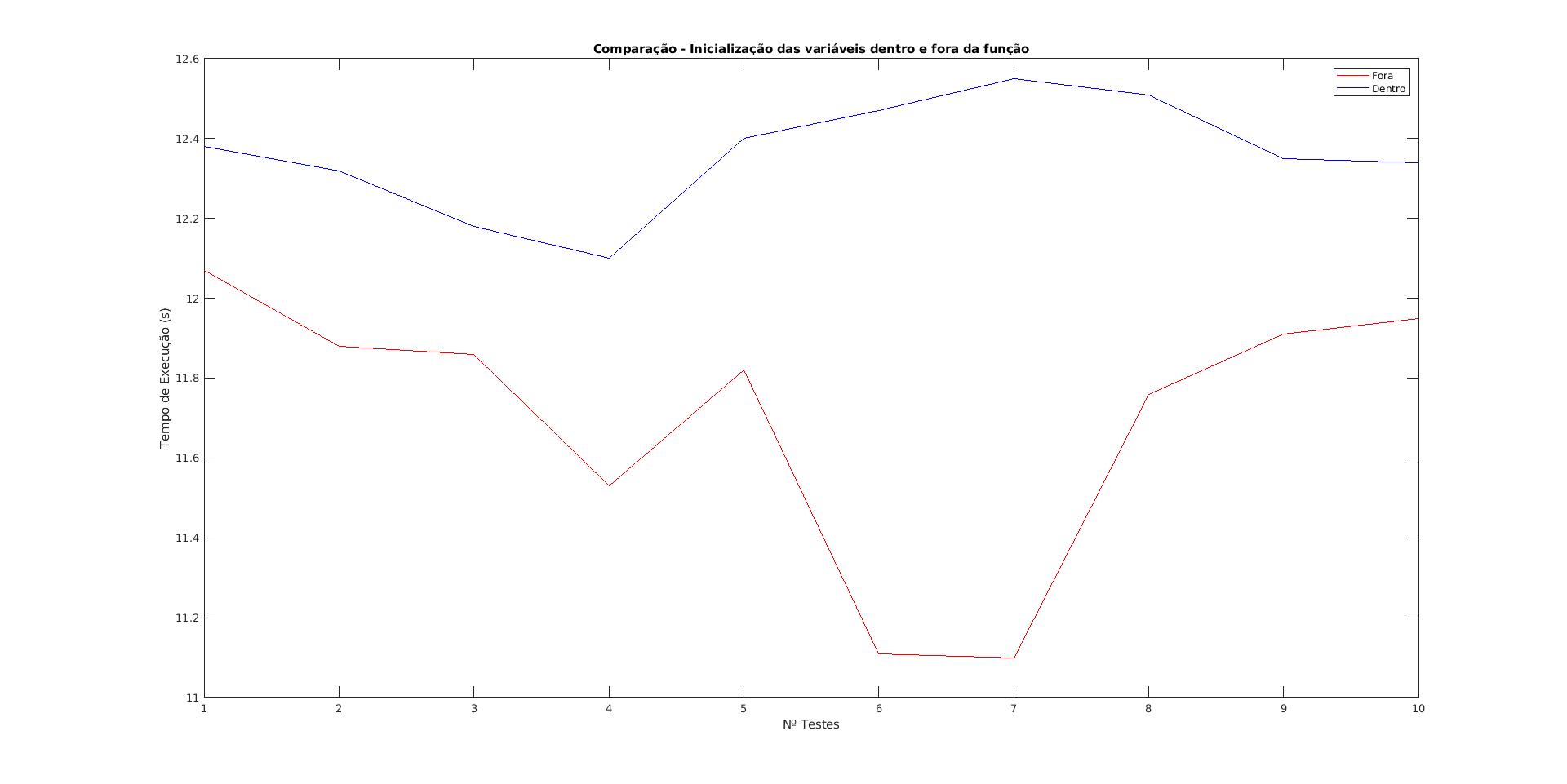
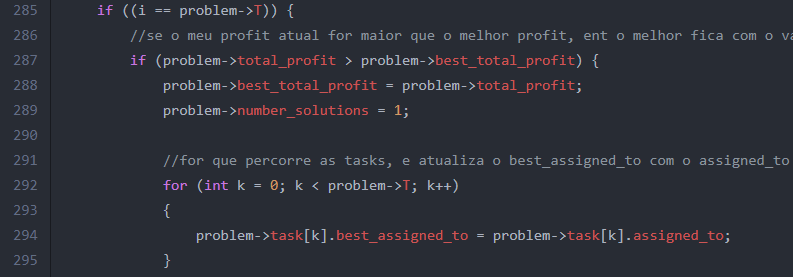
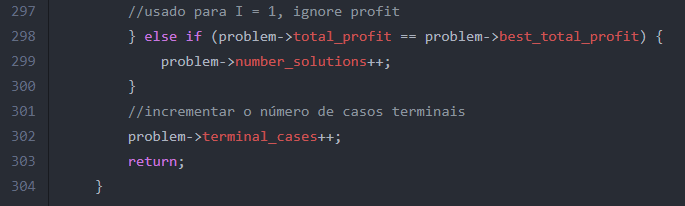


Figura - Comparação tempos de execução

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*, 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. FALAR COM A EVA AMANHA SOBRE O NUMBER\_SOLUCTIONS*.* 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.



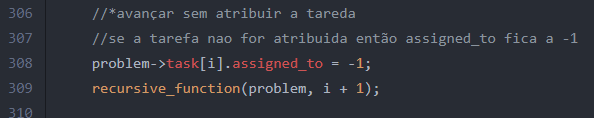
Figura

Ainda, dentro deste caso especial, temos uma expressão condicional, 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”, FALAR COM A EVA AMANHA . Antes de sairmos deste bloco de código incrementamos a variável *terminal\_cases* unitariamente, que servirá para contar o número de casos viáveis totais.

Figura

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 avança.

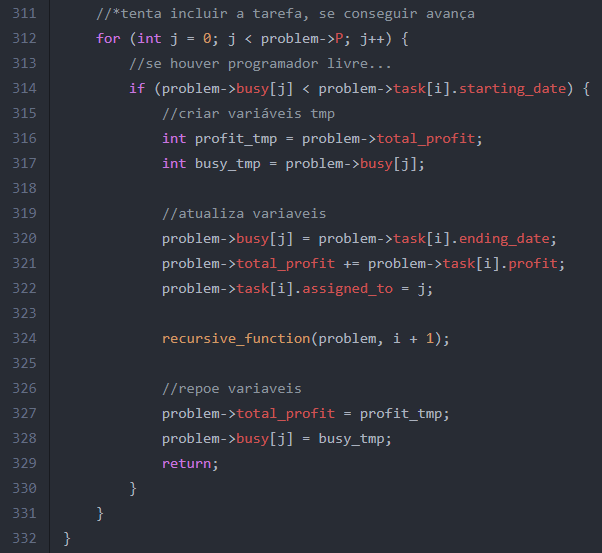
Para avançar sem atribuir a tarefa, basta chamar a própria função, no nível seguinte, *i+1.* Assim, estamos a não atribuir esta *Tarefa*, deste modo, vamos atualizar o *assigned\_to* do nível em que nos encontramos para -1, pois como já referido anteriormente, a tarefa não será atribuída a nenhum programador.



Figura

Para tentar incluir a *Tarefa*, 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”.



Figura

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 *tas*k 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

??TENHO ALGUMA DUVIDAS EM SABER SE USAMOS DIVIDE AND CONQUER??

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*, pois a nossa questão é de caráter exponencial e exige analisar bastantes possibilidades. Esta técnica algorítmica, permite-nos “retirar” possibilidades combinatórias da nossa arvore, 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 - Branch & Bound

Ora, a técnica *Divide & Conquer* 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 é bastante favorável pois as frações do nosso problema são independentes (as *tasks* não dependem umas das outras), este método agiliza a combinação das soluções, e assim obtemos um algoritmo eficiente e simples.

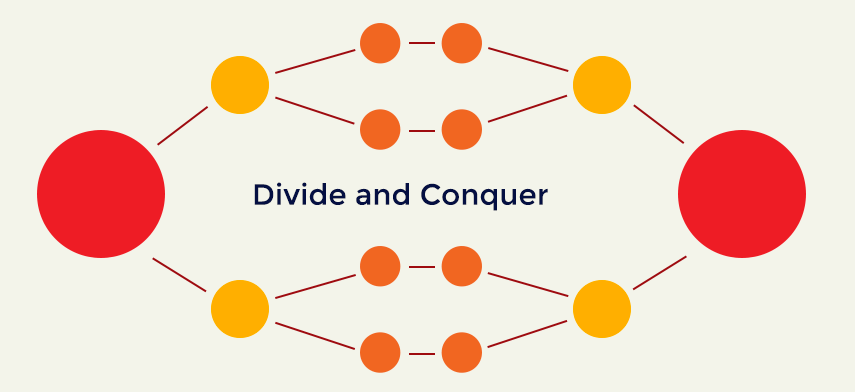


Figura - Divide & Conquer (Leitura feita da esquerda para a direita)

Seguidamente, a técnica *Dynamic Programing* 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 implementação reduz o tempo e a complexidade do problema, que em vez de ser exponencial passa a ser linear. No entanto, esta não é a melhor implementação para resolver a nossa questão, visto que esta funciona melhor quando as frações do problema são dependentes umas das outras, o que não é o nosso caso. Ou seja, estaríamos a guardar resultados que não iriamos utilizar mais tarde o que resulta num maior gasto de memoria.

## 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. Achamos por bem não incluir este código no relatório pois em nada contribuiria para o mesmo, porém pode ser consultado [aqui](https://github.com/TheScorpoi/AED_Trabalho01/tree/main/TRABALHO01_AED_JAVA).

# **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, implementamos código em Java.

Após o job\_selection\_do\_all.bash ter terminado, executamos o script *extract\_data.bash* (Fig. 4), disponibilizado na página online da unidade curricular, e com algumas modificações feitas por nós, to que toca à localização onde os ficheiros serão criados. 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.



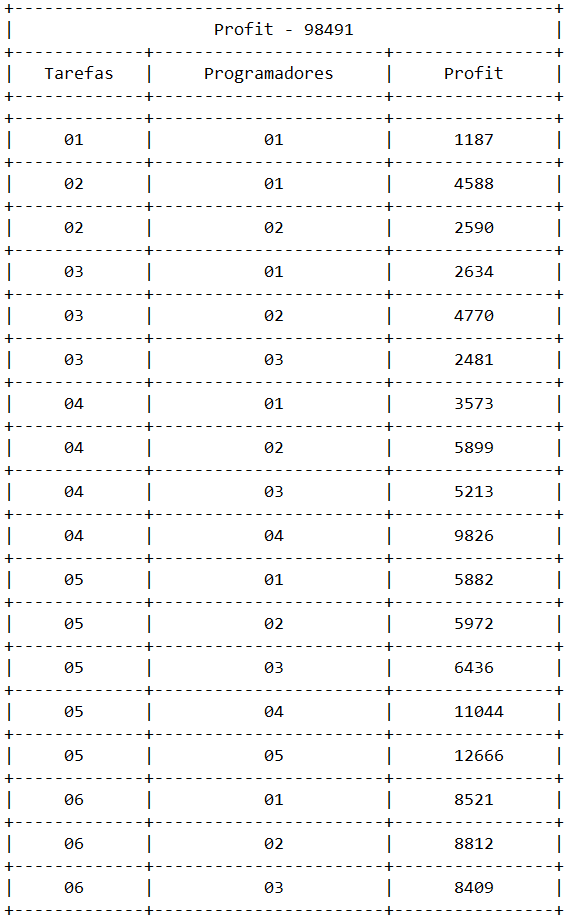
Figura

## 4.1 - Tabelas

No que toca à criação das tabelas, decidimos fazer uma [implementação em Java](https://github.com/TheScorpoi/AED_Trabalho01/blob/main/Tabelas.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.

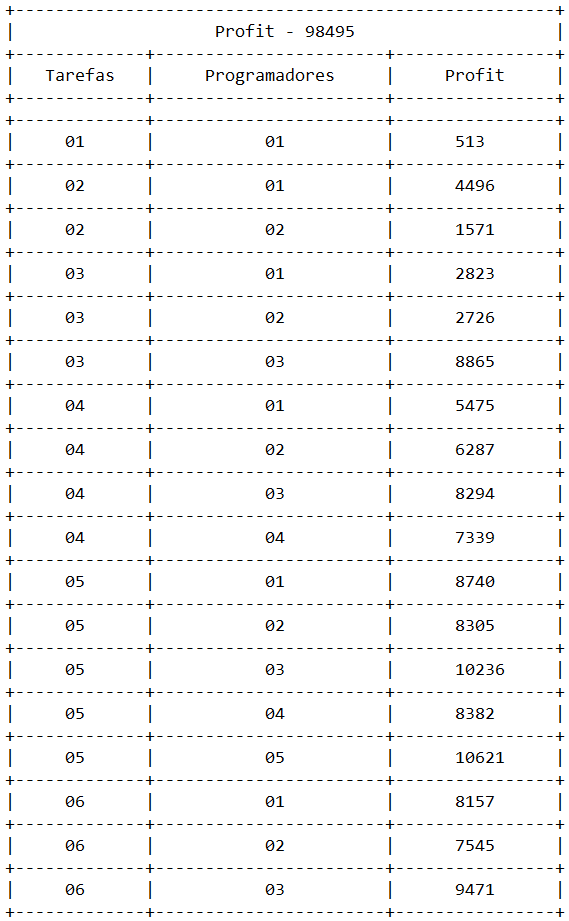
Como conseguimos resolver o problema até 40 *Tarefas e 8* *Programadores*, temos um total de 291 tarefas feitas, o que implica que a tabela seja muito comprida, impossibilitando assim que a mesma fique completa neste relatório. As tabelas completas podem ser consultadas [aqui](https://github.com/TheScorpoi/AED_Trabalho01/tree/main/Resultados/Tabelas).

A tabela seguinte (Fig. 5), é representativa do elemento do grupo com o número mecanográfico 98491.



Figura

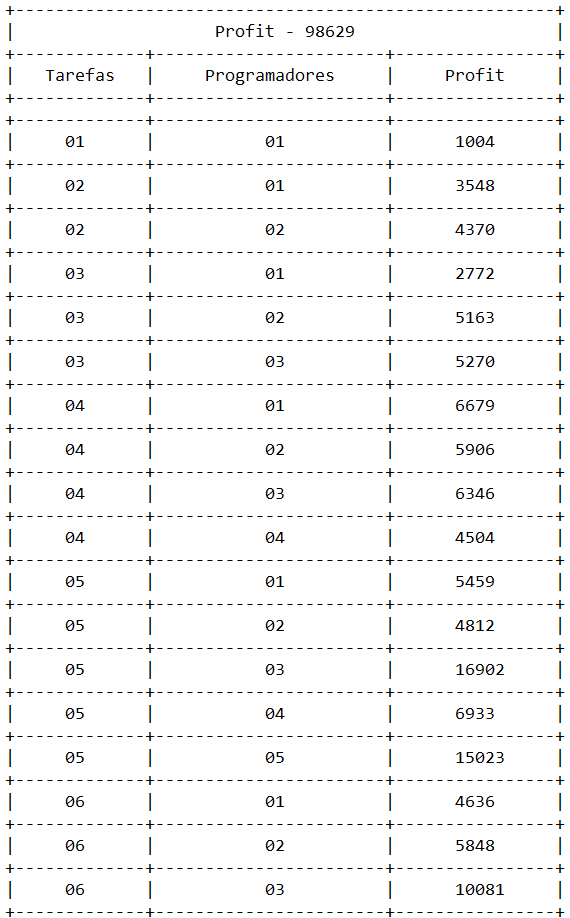
A tabela seguinte (Fig. 6), é representativa do elemento do grupo com o número mecanográfico 98495.



Figura

­­

A seguinte tabela (Fig. 7), representa os *profits* obtidos pelo elemento do grupo com o número mecanográfico 98629.



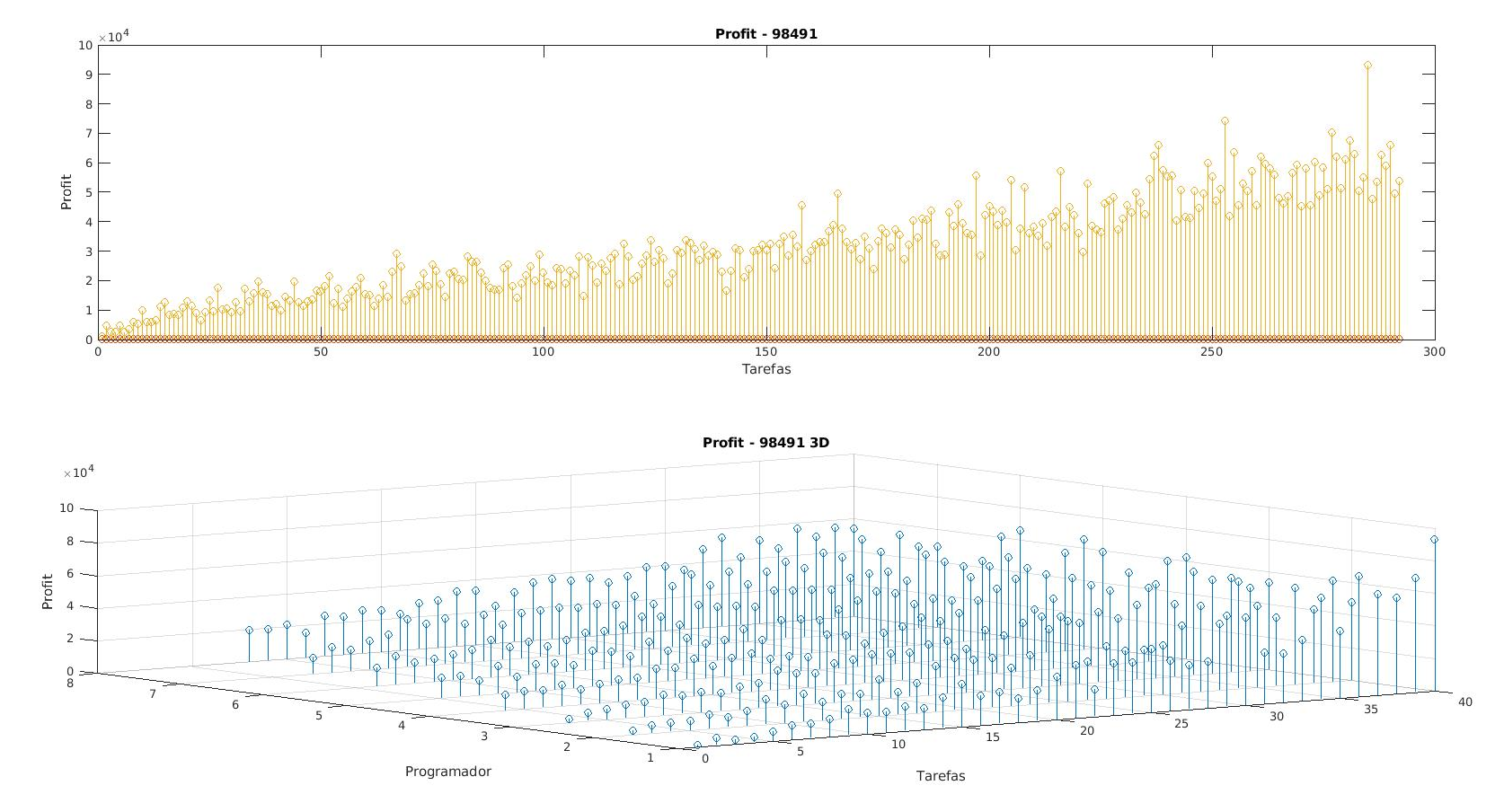
Figura

## 4.2 – Gráficos

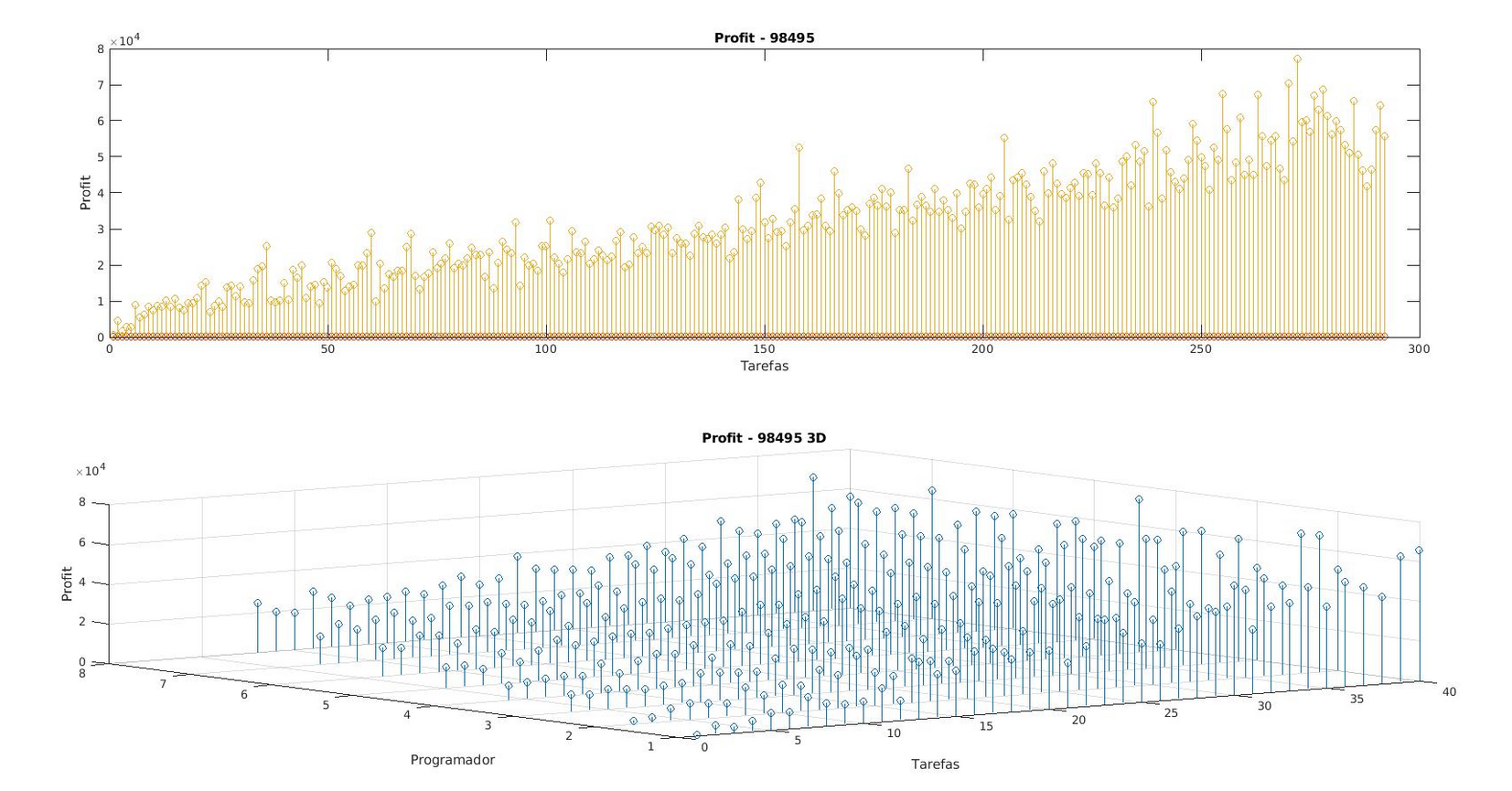
Em relação aos gráficos criados, criamos gráficos com e para diversos propósitos, para demonstrar o profit que cada elemento do grupo alcançou, para ilustrar os tempos de execução por parte de cada elemento, e ainda gráficos para a parte de se ignorar o profit. Todos os gráficos foram criados com recurso 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](https://github.com/TheScorpoi/AED_Trabalho01/tree/main/Resultados/MatLab), ou então no ponto 5 do relatório – [Apêndice](#_5-Apêndice).

### 4.2.1 – Profit

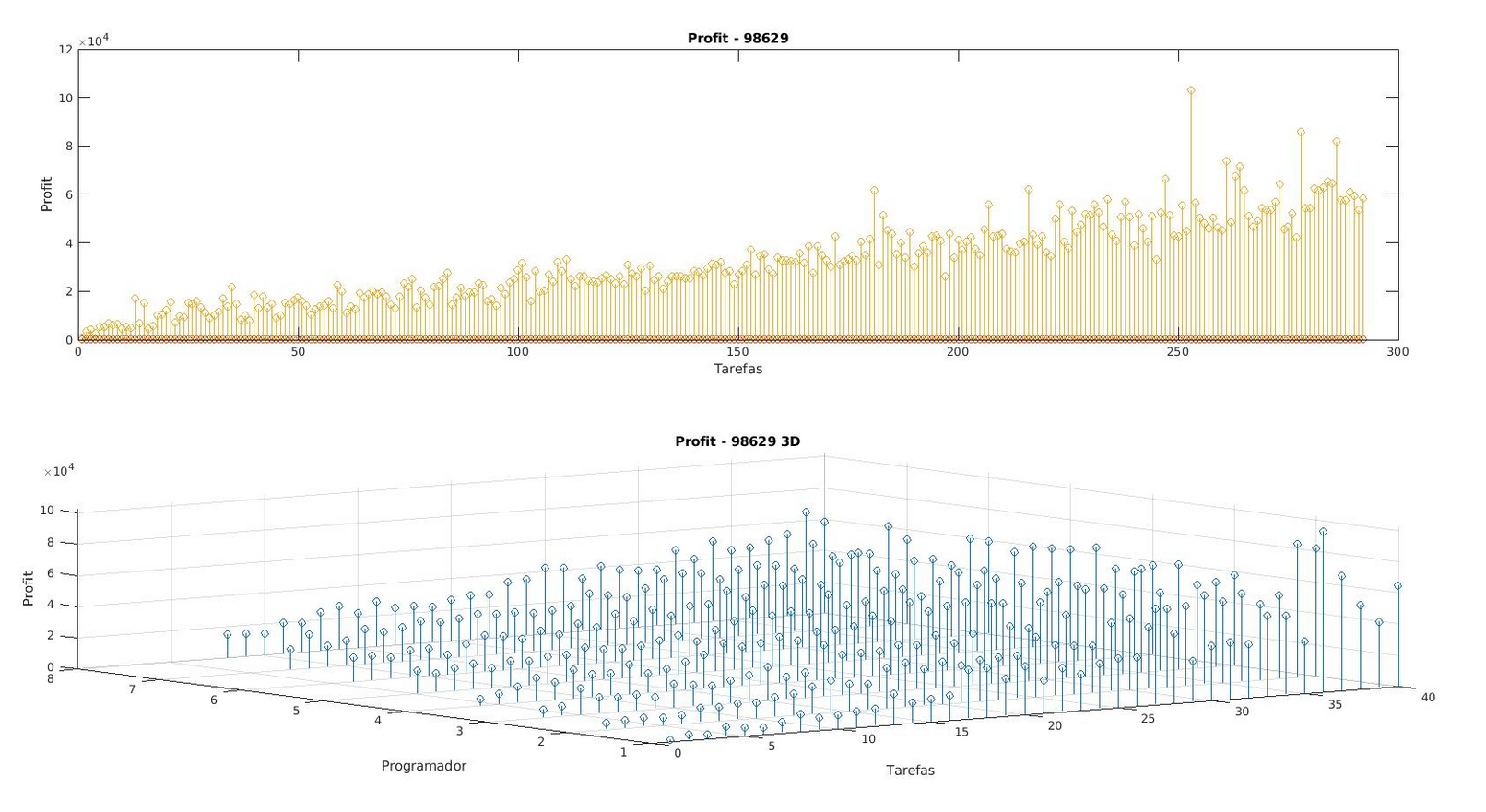
Nas três figuras anteriores (Fig. 8 ; Fig. 9 ; Fig. 10 ) estão representados os gráficos e os gráficos em 3D com os resultados dos profits com o número de tarefas a variar entre 1 e 40 e o número de programadores a variar entre 1 e 8, sendo que, a primeira figura representa os resultados para o número mecanográfico 98491, a segunda figura representas os resultados para o número mecanográfico 98495 e a terceira figura representa os gráficos para o número mecanográfico 98629.



Figura

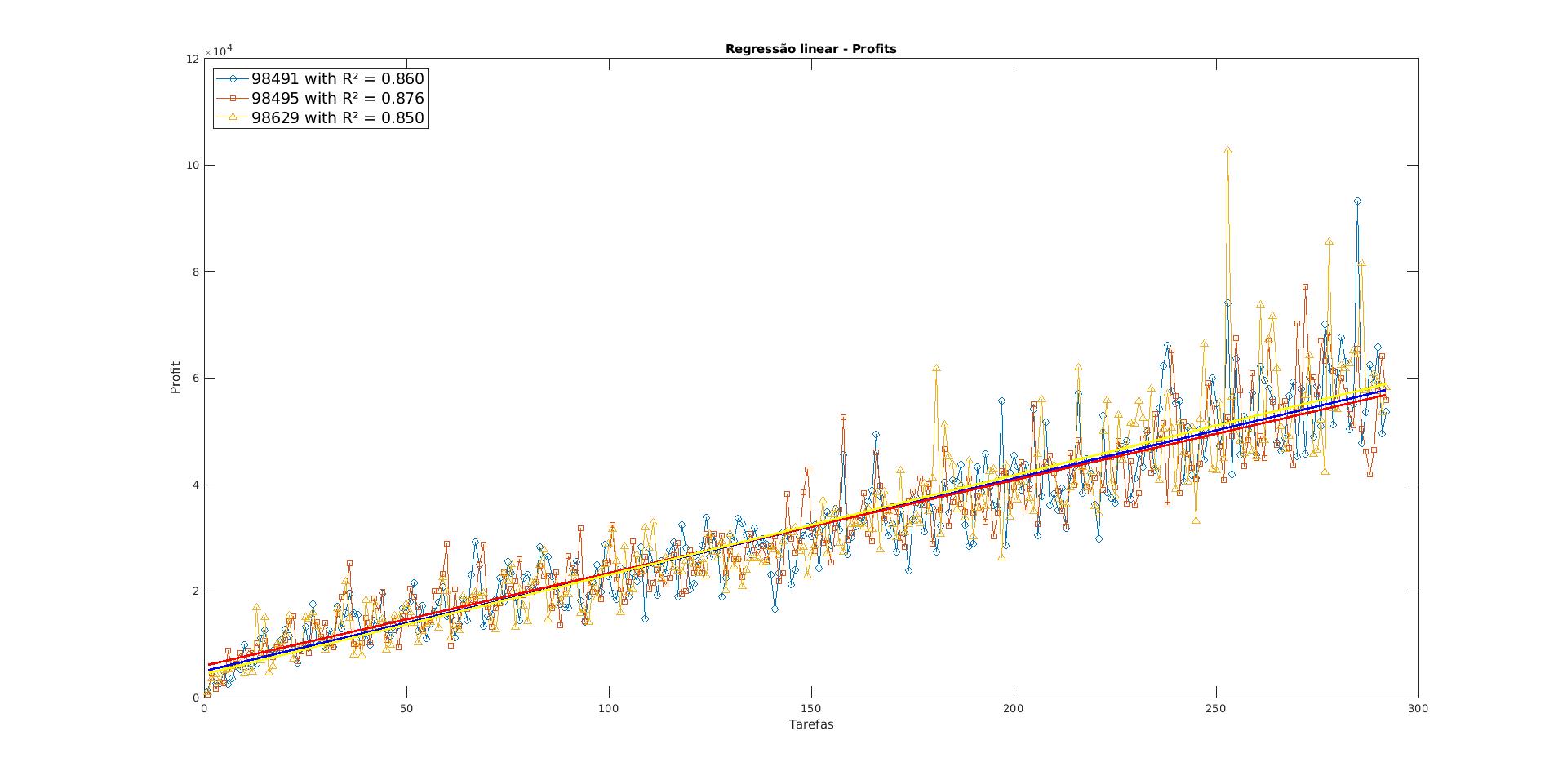


Figura



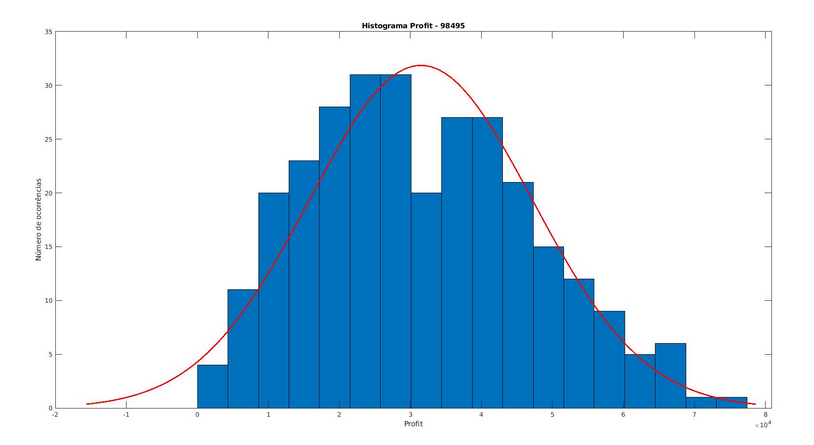
Figura

Figura

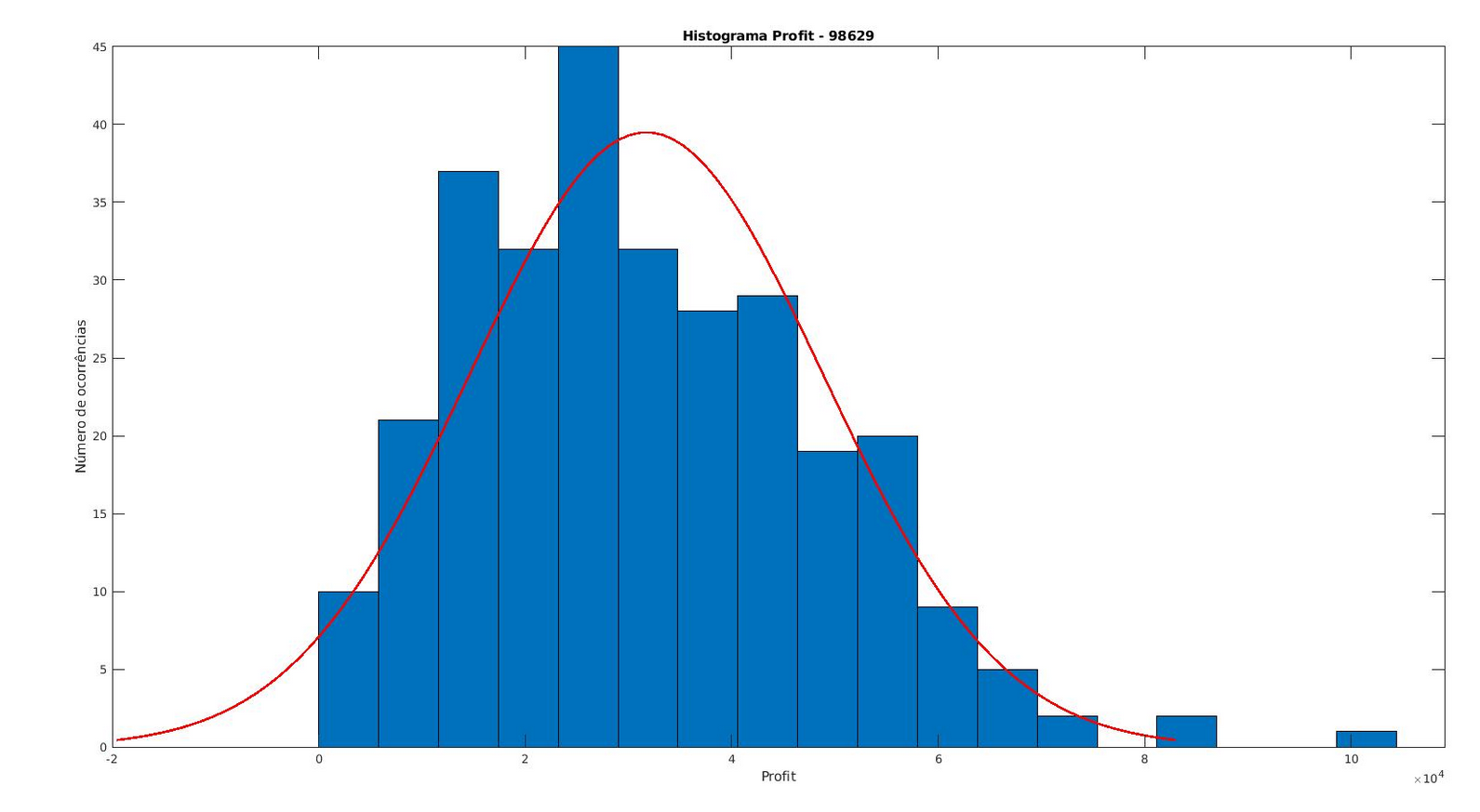


### 4.2.2 – Histogramas Profit

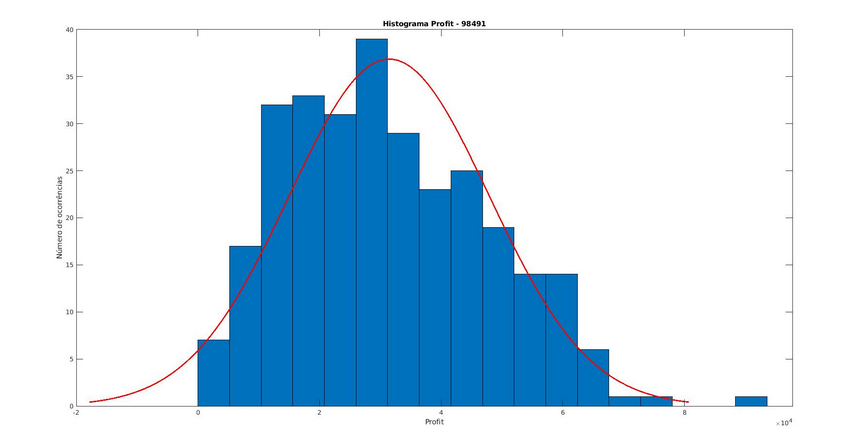
Ora, neste ponto vamos contar o número de tarefas validas. E, contruir histogramas sobre o número de ocorrências de cada *profit*, e retirar conclusões.



Figura



Figura



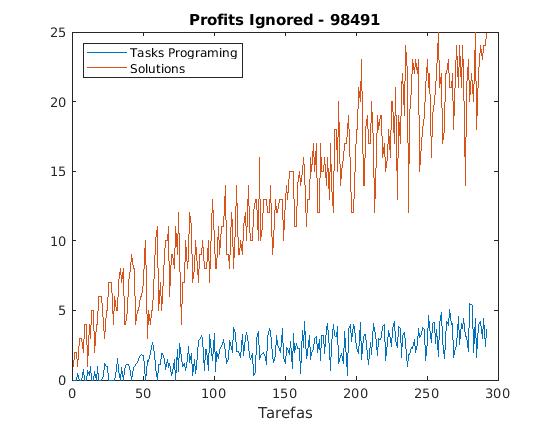
Figura

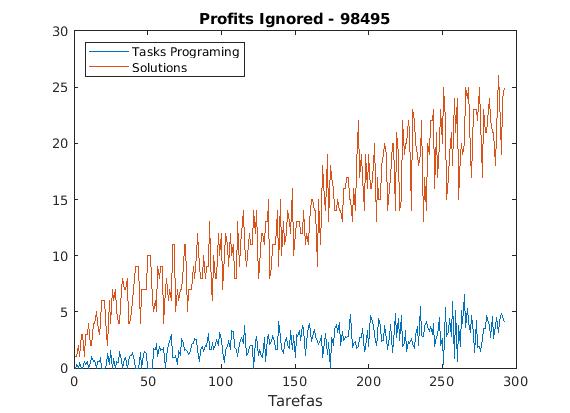
Nas três figuras anteriores (Fig. 12; Fig. 13; Fig. 14) estão representados os histogramas com os resultados dos profits com o número de tarefas a variar entre 1 e 40 e o número de programadores a variar entre 1 e 8, sendo que, a primeira figura representa os resultados para o número mecanográfico 98495, a segunda figura representa os resultados para o número mecanográfico 98629 e a terceira figura representa os gráficos para o número mecanográfico 98491.

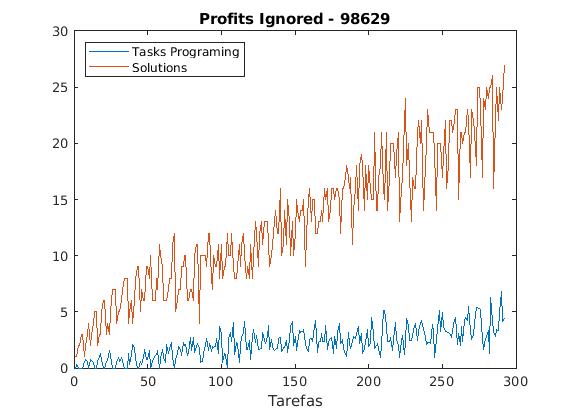
|  |  |
| --- | --- |
|  |  |
|  |  |

### 4.2.3 – Ignorar Profits

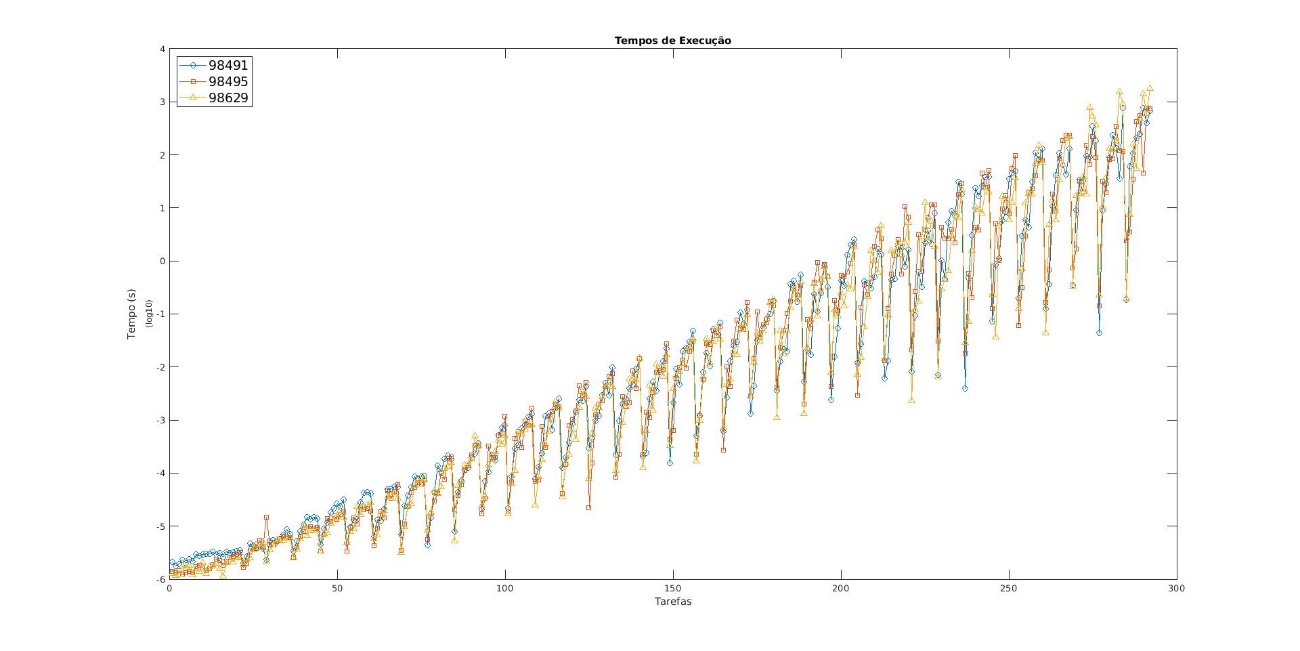
Decidimos por bem, correr a nossa implementação para quando os profits são ignorados, visto que durante a criação da função recursiva, fizemos código para tratar esta eventualidade, criamos os seguintes gráficos. Para extrair a informação, criamos o *script* *bash* *extract\_without\_profit.sh,* que podem ser consultados [aqui](#_5.3_–_Ficheiro), onde retiramos desta forma as *Programing Taks* e os *Terminal Cases*.



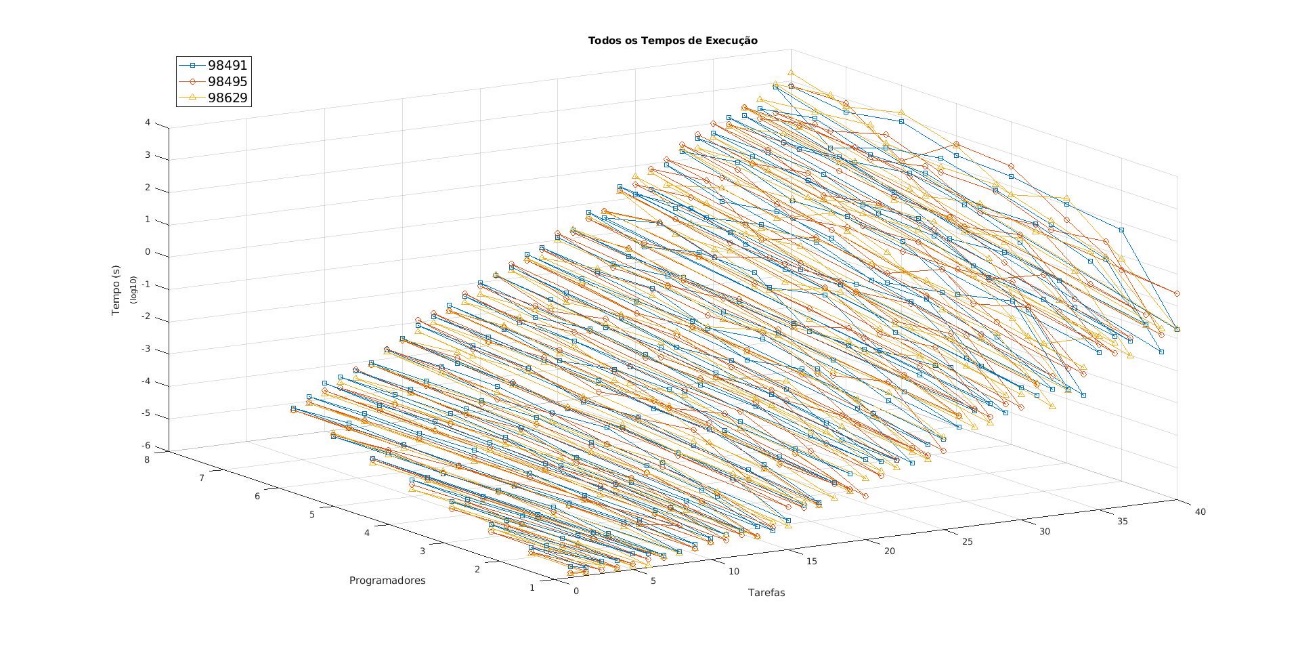




### 4.2.4 – Tempos de Execução



Figura



Figura

Nas figuras 15 e 16 estão representados o gráfico com os tempos de execução e o gráfico 3D com os tempos de execução, respetivamente.

# **5 - Apêndice**

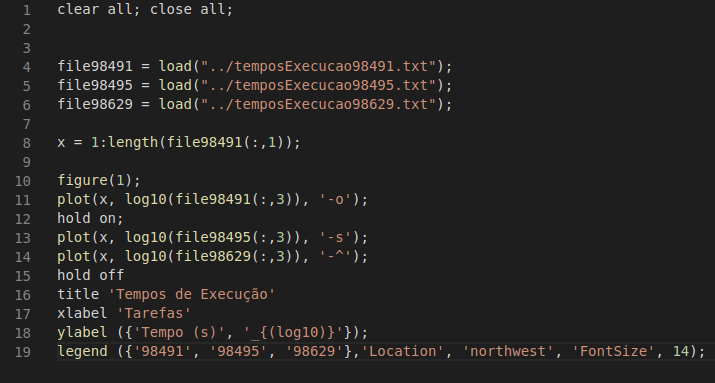
## 5.1 - Tabelas



Figura

O código java representado na figura anterior (Fig. 17) é usado para gerar as tabelas com os profits para os diferentes números de programadores, de tarefas e para os diferentes números mecanográficos.

## 5.2 - Matlab



Figura

Na figura anterior (Fig. 18) está representado o código em MatLab que é usado para gerar os gráficos com os tempos de execução.



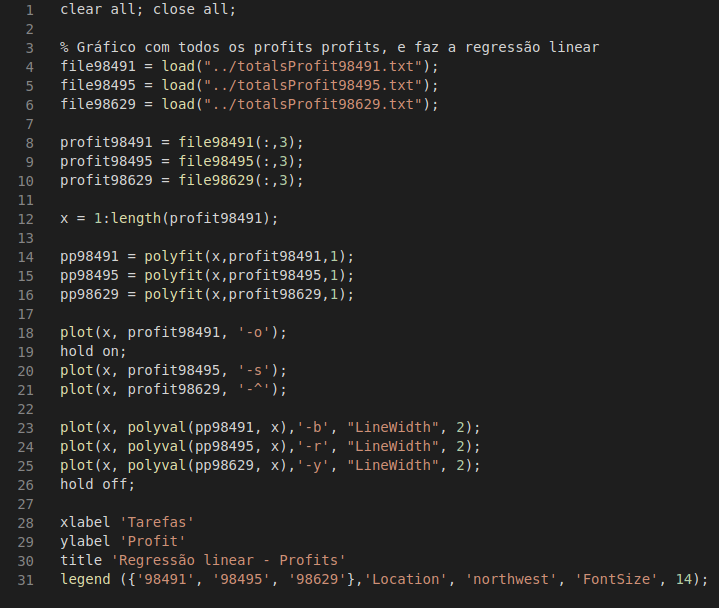
Figura

Na figura anteriormente apresentada (Fig. 19) está representado o código MatLab usado para a criação dos gráficos 3D com os tempos de execução.



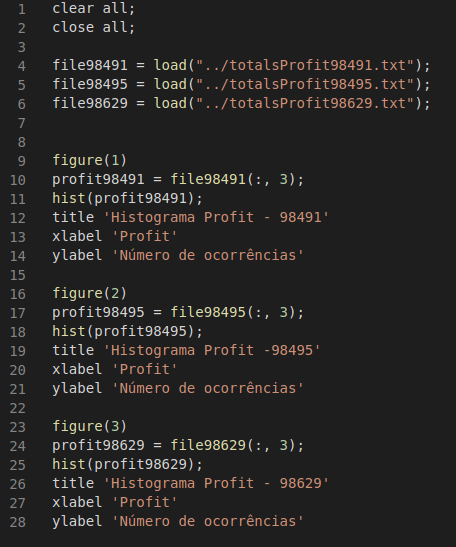
Figura

Na figura 20 está representado o código que coloca todos os profits a 1 e analisa qual é a melhor solução.



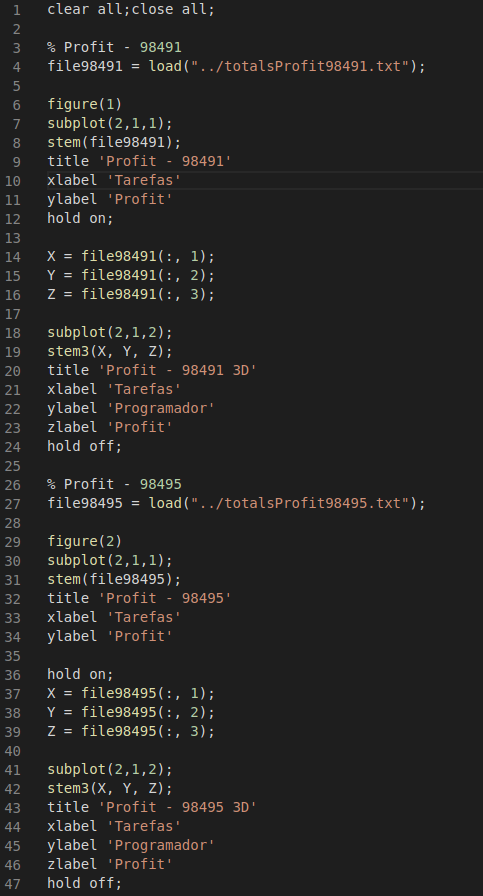
Figura

Nesta figura (Fig. 21) está a ser mostrado o código em MatLab usado para criar um gráfico com a regressão linear dos profits dos números mecanográficos 98491, 98495 e 98629.

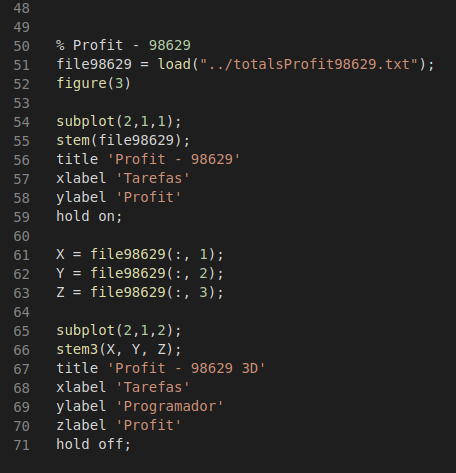


Figura

O código utilizado para criar os histogramas com os profits dos números mecanográficos 98491, 98495 e 98629 está representado na figura anterior (Fig. 22).



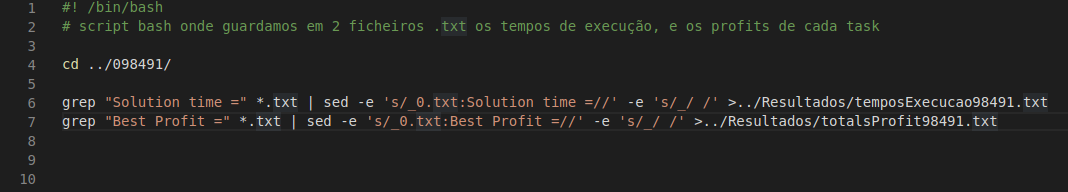
Figura



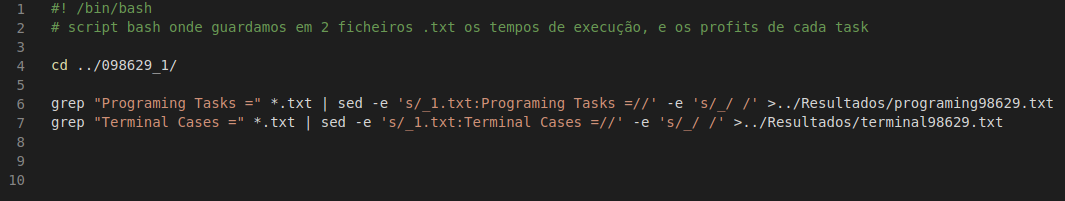
Figura

O código MatLab representado nas duas figuras anteriores (Fig. 23; Fig. 24) é utilizado para a criação dos gráficos 3D com os profits dos números mecanográficos 98491, 98495, 98629.

## 5.3 – Ficheiro BASH



Figura



Figura

# **6 - Conclusão**

# **7 - Bibliografia**

Silva, Tomás Oliveira e. Lecture notes: Algorithms and Data Structute (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]