

Estrutura de Dados Avançadas

Engenharia De Sistemas
Informáticos

Nome: Henrique da Silva Araújo Neto **Nº** 16626

Escola Superior De Tecnologia
Instituto Politécnico do Cávado e Do Ave

Índice

Índice	1
Introdução	2
Propósitos e Objetivos.....	4
Estrutura de dados	4
Fase 1:	4
Testes Realizados.....	5
Fase 1:	5
1. Definição de uma estrutura de dados dinâmica para a representação de um <i>job</i> com um conjunto finito de <i>n</i> operações:	5
2. Armazenamento/leitura de ficheiro de texto com representação de um <i>job</i> ;	6
3. Inserção de uma nova operação:	8
4. Remoção de uma determinada operação:.....	9
5. Alteração de uma determinada operação:	10
6. Determinação da quantidade mínima de unidades de tempo necessárias para completar o <i>job</i> e listagem das respetivas operações:	11
7. Determinação da quantidade máxima de unidades de tempo necessárias para completar o <i>job</i> e listagem das respetivas operações:	12
8. Determinação da quantidade média de unidades de tempo necessárias para completar uma operação, considerando todas as alternativas possíveis:	14
Funções de ajuda.....	15
Git	17
Execução.....	17
Conclusão	18
Bibliografia	19

Introdução

Para mostrar os conhecimentos adquiridos relativos a definição e manipulação de estruturas de dados dinâmicas na linguagem de programação C, foi pedido aos alunos de Engenharia de Sistemas Informáticos (LESI) o desenvolvimento de uma solução digital para um problema de escalonamento denominado *Flexible Job Shop Problem (FJSSP)*. A solução deverá permitir gerar uma proposta de escalonamento para a produção de um produto envolvendo várias operações e a utilização de várias máquinas, minimizando o tempo as unidades de tempo necessário na sua produção (*makespan*).

Para a realização da solução, esta foi dividida em duas fases:

Fase 1:

1. Definição de uma estrutura de dados dinâmica para a representação de um job com um conjunto finito de n operações;
2. Armazenamento/leitura de ficheiro de texto com representação de um job;
3. Inserção de uma nova operação;
4. Remoção de uma determinada operação;
5. Alteração de uma determinada operação;
6. Determinação da quantidade mínima de unidades de tempo necessárias para completar o job e listagem das respetivas operações;
7. Determinação da quantidade máxima de unidades de tempo necessárias para completar o job e listagem das respetivas operações;
8. Determinação da quantidade média de unidades de tempo necessárias para completar uma operação, considerando todas as alternativas possíveis.

Fase 2:

1. Definição de uma estrutura de dados dinâmica para representação de um conjunto finito de m jobs associando a cada job um determinando conjunto finito de operações;
2. Armazenamento/leitura de ficheiro de texto com representação de um process plan;
3. Inserção de um novo job;
4. Remoção de um job;
5. Inserção de uma nova operação num job;
6. Remoção de uma determinada operação de um job;
7. Edição das operações associadas a um job;
8. Cálculo de uma proposta de escalonamento para o problema FJSSP (obrigatoriamente limitado a um tempo máximo de processamento configurável), apresentando a distribuição das operações pelas várias máquinas, minimizando o makespan (unidades de tempo necessárias para a realização de todos os jobs). A proposta de escalonamento deverá ser exportada para um ficheiro de texto possibilitando uma interpretação intuitiva;
9. Representação de diferentes process plan (variando a quantidade de máquinas disponíveis, quantidade de job, e sequência de operações, etc) associando as respetivas propostas de escalonamento.

Propósitos e Objetivos

O presente trabalho têm como propósito mostrar os conhecimentos adquiridos relativos a definição e manipulação de estruturas de dados dinâmicas na linguagem de programação C, como também mostrar os conhecimentos na utilização de ferramentas como: Git e DoxyGen.

Estrutura de dados

Fase 1:

Para a realização da fase 1 do trabalho foi utilizada a estrutura de dados linear e dinâmica, **lista ligada**.

A **lista ligada** é composta por vários “nós” que estão interligados através de apontadores, ou seja, cada “nó” possui um apontador que aponta para o endereço de memória do próximo “nó”.



Figura 1: Representação de uma lista ligada

Testes Realizados

Fase 1:

1. Definição de uma estrutura de dados dinâmica para a representação de um *job* com um conjunto finito de *n* operações:

Para representar um *job*, foi criada uma lista ligada de máquinas:

```
typedef struct machine {
    int nOperation;
    int nMachine;
    int vTime;
    struct machine *proximo;
} ListMachines;
```

Figura 2: Representação de uma máquina por uma struct

nOperation: Operação em que a máquina está inserida;

nMachine: Número da máquina;

vTime: Unidades de tempo da máquina;

struct machine: apontador para a próxima máquina.

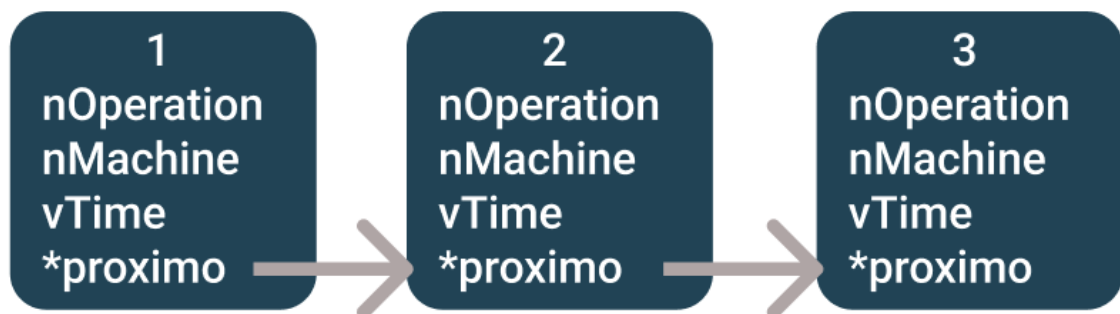


Figura 3: Lista ligada com os nós de uma máquina

2. Armazenamento/leitura de ficheiro de texto com representação de um job;

Para realizar o armazenamento e leitura de um ficheiro de texto foram criadas três funções:

- *saveJob()* : Salva as operações de um job no ficheiro txt;
- *saveJobFromList()* : Salva no ficheiro de texto todas as operações do job que está guardada na memória;
- *readJob()*: Realiza a leitura de um ficheiro txt com as operações relacionadas a um job e salva na memória.

saveJob() :

```
void saveJob(int nOperation, int nMachine, int vTime) {  
  
    if (verifyIfFileExist("job.txt") == FALSE) {  
  
        FILE *f = fopen("job.txt", "a");  
  
        fclose(f);  
  
    }  
  
    if (verifyIfFileExist("job.txt")) {  
  
        FILE *f = fopen("job.txt", "a");  
  
        fprintf(f, "(%d,%d,%d)\n", nOperation, nMachine, vTime);  
  
        fclose(f);  
  
    }  
  
}
```

Figura 4: função para salvar operações de um job

Esta função recebe o número da operação, o número da máquina e o seu tempo, que foram inseridos pelo utilizador.

Caso já exista um ficheiro com a representação do job, a função só salva os novos dados no ficheiro, em caso do ficheiro não existir, este é criado e de seguida é adicionada a nova operação no ficheiro.

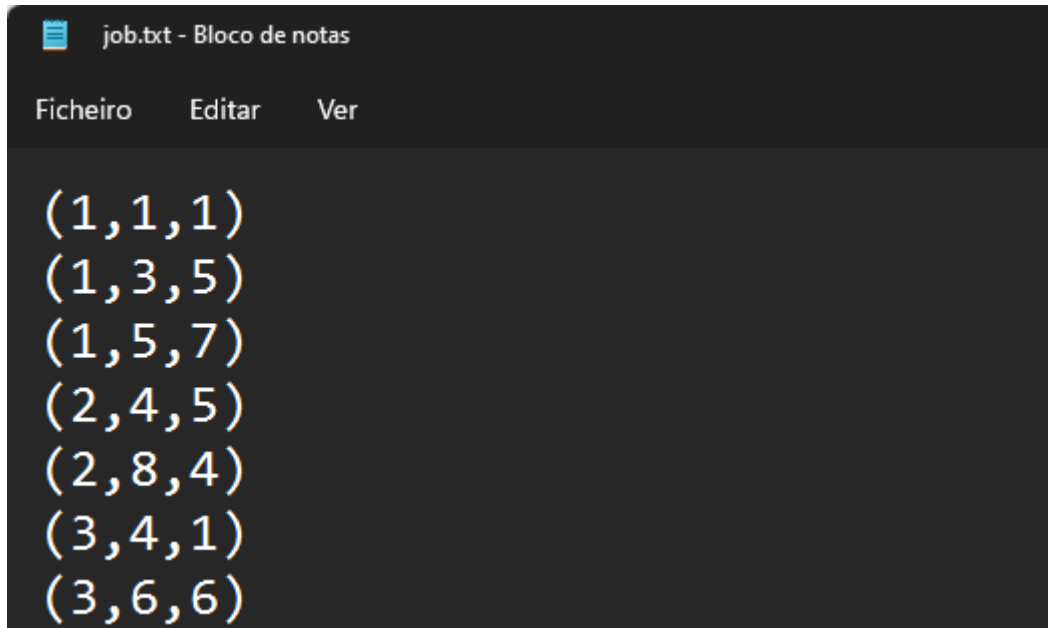


Figura 5: Ficheiro txt com a representação de um job

(a, b, c):

a: Número da operação;

b: Número da máquina;

c: Unidades de tempo na máquina;

3. Inserção de uma nova operação:

Para realizar a inserção de uma nova operação, foram utilizadas duas funções:

- *insertAtBegin()*: Adiciona uma nova operação no começo da lista;
- *newOperationInputs()*: Recebe todos os inputs do utilizador; relativos a uma nova operação.

insertAtBegin():

```
ListMachines *insertAtBegin(int nOperation, int nMachine, int vTime, ListMachines *list) {
    ListMachines *new = (ListMachines *)malloc(sizeof(ListMachines));

    if (new != NULL) {
        new->nOperation = nOperation;
        new->nMachine = nMachine;
        new->vTime = vTime;
        new->proximo = list;

        return(new);
    } else {
        return(list);
    }
}
```

Figura 6: Função para inserir uma nova máquina no começo da lista

Depois de receber o número da operação, número da máquina e unidades de tempo da máquina, a função retorna o novo nó que será inserido no começo da lista de máquinas.

newOperationInputs():

Depois de verificar e testar todos os dados recebidos pelo utilizador, esta função guarda estes dados dentro das respetivas variáveis e envia para para a função *insertAtBegin()*.

```
listMachines = insertAtBeginAndFile(nOperationInput, nMachineInput, vTimeInput, listMachines);
```

Figura 7: Chamada da função *insertAtBeginAndFile*

4. Remoção de uma determinada operação:

A função *removeOperation()* recebe como parâmetro o número da operação que será removida e com a ajuda de dois apontadores guarda o nó anterior e o nó atual para no fim poder remover e realizar a ligação para o nó seguinte.

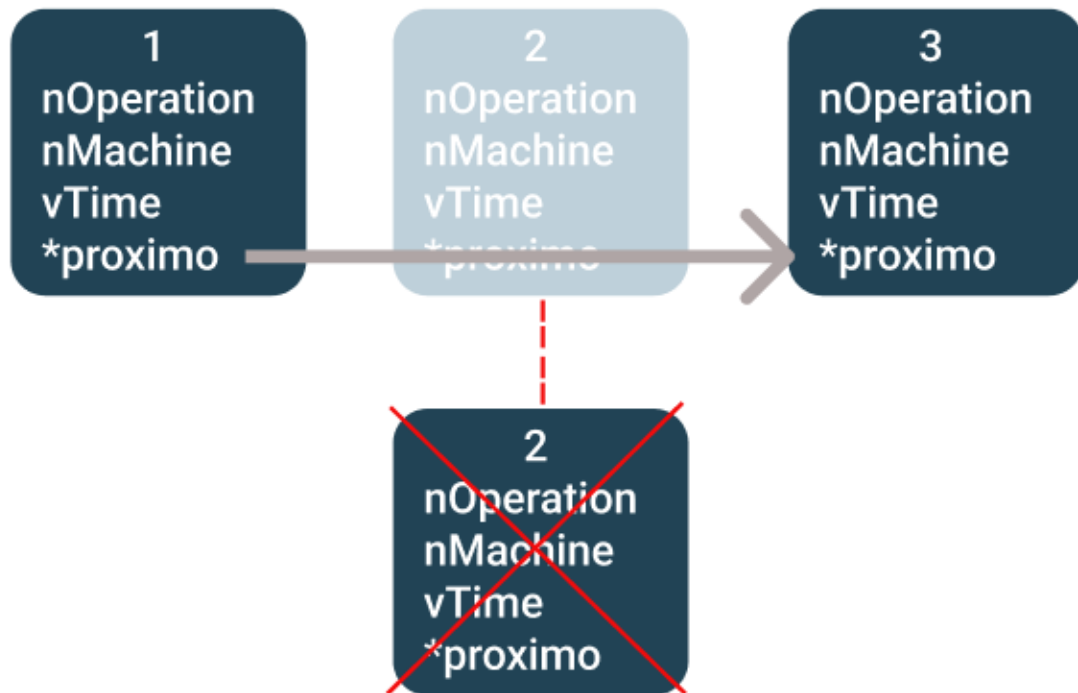


Figura 8: Representação de remoção de um nó

Como vários nós contêm o mesmo número de operação, foi utilizado uma condicional *if* que verifica qual a operação que está contida no nó atual e remove caso seja a que o utilizador inseriu.

```
if (listHead != NULL && listHead->nOperation == nOperation) {
    while (listHead != NULL && listHead->nOperation == nOperation) {
        nodeToRemove = listHead;
        listHead = listHead->proximo;
        free(nodeToRemove);
        removed = TRUE;
    }
    listMachines = listHead;
}
```

Figura 9: Condição para se remover uma operação

5. Alteração de uma determinada operação:

Através da função *editOperation()*, podemos editar as máquinas da mesma.

editOperation():

A função recebe como parâmetro o número da operação a ser editada e pergunta ao utilizador qual máquina ele deseja editar.

A edição é feita alterando os dados contidos no nó em que a máquina se encontra.

```
listHead->nMachine = newMachineNumber;
listHead->vTime = newTimeNumber;

remove("job.txt");
saveJobFromList(listMachines);
```

Figura 10: Edição de uma operação

6. Determinação da quantidade mínima de unidades de tempo necessárias para completar o job e listagem das respectivas operações:

Com a função *minimumAmountOfTime()* conseguimos percorrer toda a lista de máquinas e criar uma nova somente com as máquinas de menor tempo.

minimumAmountOfTime():

Com a ajuda de dois apontadores, a função percorre toda a lista de máquinas e verifica quais os nós que devem ser adicionados a nova lista.

```
if (listHead->proximo != NULL && listHead->nOperation == listHead->proximo->nOperation) {
    if (aux->vTime >= listHead->proximo->vTime) {
        aux = listHead->proximo;
    }
} else {
    //Guardar na lista o aux
    newList = insertAtBegin(aux->nOperation, aux->nMachine, aux->vTime, newList);
    aux = listHead->proximo;
}

listHead = listHead->proximo;
```

Figura 11: Verificação de um nó

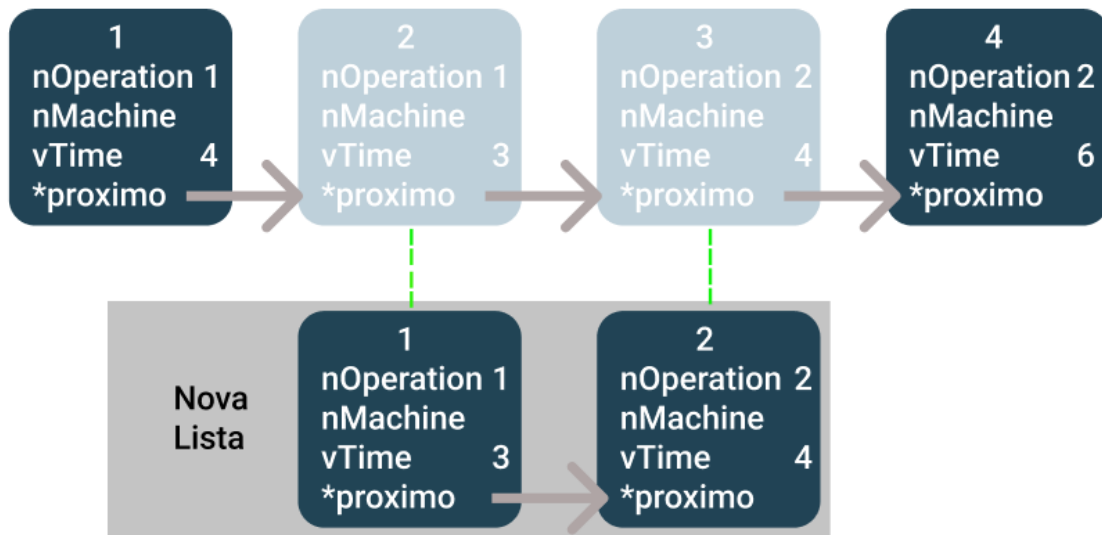


Figura 12: Representação da criação de uma nova lista a partir de outra

No final é chamada a função *sumMachineTime()* que recebe uma lista de máquinas e retorna a soma de todas as unidades de tempo.

```
int sumMachineTime(ListMachines *list) {
    ListMachines *listHead = list;
    int sum;

    while (listHead != NULL) {
        sum += listHead->vTime;
        listHead = listHead->proximo;
    }

    return sum;
}
```

Figura 13: Soma de todas as unidades de tempo

7. Determinação da quantidade máxima de unidades de tempo necessárias para completar o job e listagem das respectivas operações:

Com a função *maximumAmountOfTime()* conseguimos percorrer toda a lista de máquinas e criar uma nova somente com as máquinas de maior tempo.

maximumAmountOfTime():

Com a ajuda de dois apontadores, a função percorre toda a lista de máquinas e verifica quais os nós que devem ser adicionados a nova lista.

```
if (listHead->proximo != NULL && listHead->nOperation == listHead->proximo->nOperation) {
    if (aux->vTime <= listHead->proximo->vTime) {
        aux = listHead->proximo;
    }
} else {
    // Guardar na lista o aux
    newList = insertAtBegin(aux->nOperation, aux->nMachine, aux->vTime, newList);
    aux = listHead->proximo;
}

listHead = listHead->proximo;
```

Figura 14: Verificação de um nó

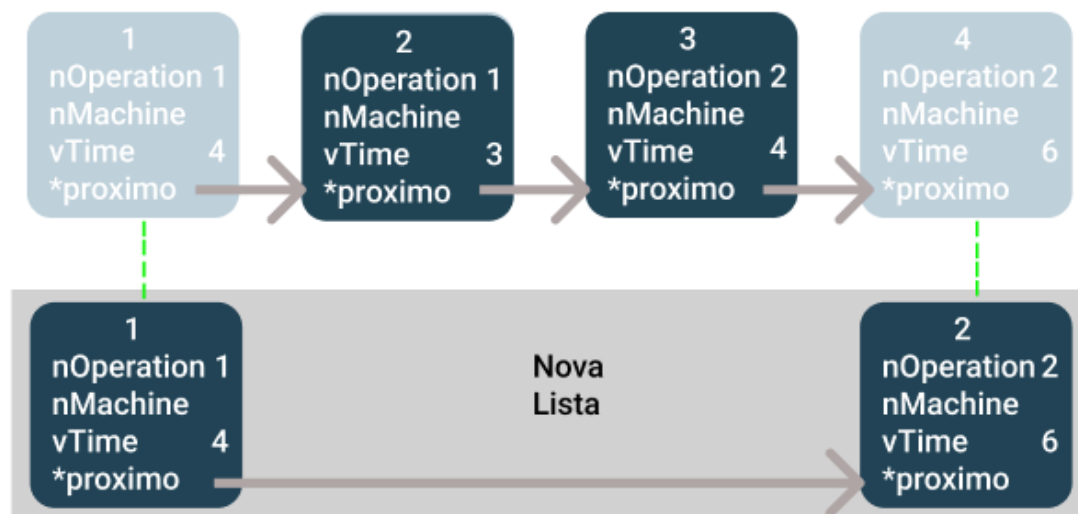


Figura 15: Representação da criação de uma nova lista a partir de outra

No final é chamada a função `sumMachineTime()` que recebe uma lista de máquinas e retorna a soma de todas as unidades de tempo.

```
int sumMachineTime(ListMachines *list) {
    ListMachines *listHead = list;
    int sum;

    while (listHead != NULL) {
        sum += listHead->vTime;
        listHead = listHead->proximo;
    }

    return sum;
}
```

Figura 16: Soma de todas as unidades de tempo

8. Determinação da quantidade média de unidades de tempo necessárias para completar uma operação, considerando todas as alternativas possíveis:

A função *averageOperationTime()* percorre a lista de máquinas e cria uma nova com a operação e sua media de unidade de tempo.

A nova lista criada, têm como base uma nova estrutura representando a média de unidades de tempo das máquinas relativas a operação, *averageOp*:

```
typedef struct averageOp {
    int nOperation;
    float vTime;
    struct averageOp *proximo;
} ListAverageOp;
```

Figura 17: Estrutura de uma operação com a sua média de unidades de tempo das máquinas

averageOperationTime():

Cria uma nova lista de operações a partir da lista de máquinas, guardando o número da operação e a sua respetiva unidade de tempo em um novo nó;

```
while (listHead != NULL) {
    sum += listHead->vTime;

    if (listHead->proximo != NULL && listHead->nOperation == listHead->proximo->nOperation) {
        count++;
    } else {
        count++;
        average = (float)sum / count;
        newList = insertAtBeginOp(listHead->nOperation, average, newList);
        average = 0;
        sum = 0;
        count = 0;
    }
    listHead = listHead->proximo;
}
```

Figura 18: Código para verificar nós

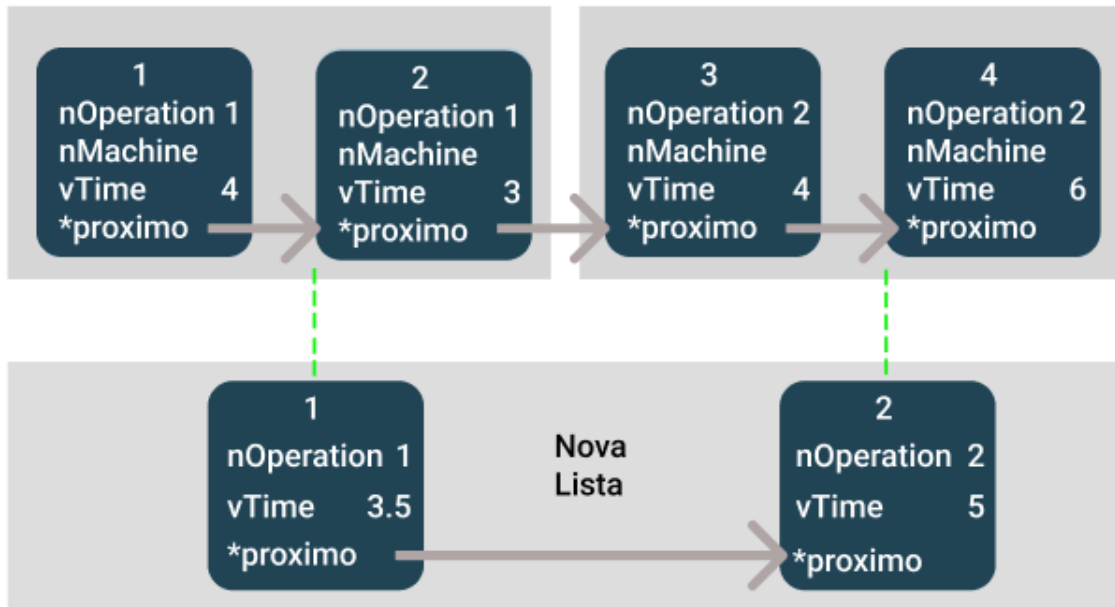


Figura 19: Representação da criação de uma nova lista a partir de outra

Funções de ajuda

Estas são as funções de ajuda, que servem para “ajudar” as funções vistas anteriormente:

- *verifyInputValues():* Recebe um inteiro e verifica se é maior que 0.

```
bool verifyInputValues(int input) {
    if (input > 0)
    {
        return TRUE;
    } else {
        return FALSE;
    }
}
```

Figura 20: Código para verificar se valor é maior que 0

- *verifyIfOperationExist()*: Recebe o número da operação e verifica se esta já existe.

```
bool verifyIfOperationExist(int operationNumber) {

    ListMachines *listHead = listMachines;

    if (listHead != NULL) {
        while(listHead != NULL) {

            if (listHead->nOperation == operationNumber) {
                return TRUE;
            }

            listHead = listHead->proximo;
        }
    } else {
        return FALSE;
    }
}
```

Figura 21: Código para verificar se operação já existe

- *VerifyIfMachineExistInOperation()*: Recebe o número da operação e da máquina e no final verifica se a máquina já foi inserida na operação.

```
bool verifyIfMachineExistInOperation(int operationNumber, int machineNumber) {

    ListMachines *listHead = listMachines;

    if (listHead != NULL){
        while(listHead != NULL) {
            if (listHead->nOperation == operationNumber) {
                if (listHead->nMachine == machineNumber) {
                    return TRUE;
                }
            }
            listHead = listHead->proximo;
        }
    } else {
        return FALSE;
    }
}
```

Figura 22: Código para verificar se máquina já existe

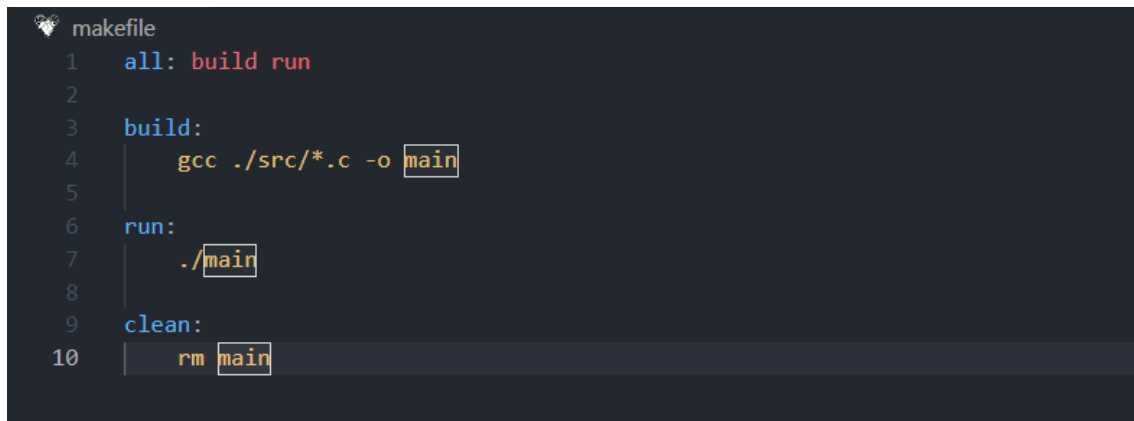
Git

Durante o desenvolvimento do programa foi utilizado o github como controle de versões:

<https://github.com/henriquearaujo93/eda-trabalho>

Execução

Para ajudar a compilar e executar o código foi utilizado o *Makefile* com as diretivas necessárias.



```
makefile
1  all: build run
2
3  build:
4      gcc ./src/*.c -o main
5
6  run:
7      ./main
8
9  clean:
10     rm main
```

Figura 23: Arquivo Makefile

all: build run -> Serve para executar a diretiva **build** e **run**;

build: -> Serve para somente compilar o código;

run: -> Serve para somente executar o código;

rm: -> Remove o executável gerado.

Conclusão

Este trabalho permite desenvolver uma solução digital para um problema de escalonamento. Consequentemente, através dele, consegui pôr em prática o que foi apresentado durante as aulas sobre estruturas de dados avançadas em linguagem C. Durante a implementação do código senti que desenvolvi as minhas habilidades em relação a estruturas de dados *listas ligadas*.

Bibliografia

Repositório Git com os códigos produzidos na aula:

<https://github.com/luferIPCA/LESI-EDA-2122.git>

Documentos de apoio fornecidos pelo professor;

Figura 1:

https://pt.wikipedia.org/wiki/Lista_ligada#/media/Ficheiro:C_language_linked_list.png