



**ESCOLA
SUPERIOR
DE TECNOLOGIA**

Instituto Politécnico do Cávado e do Ave

Licenciatura em Engenharia Informática Médica

Estruturas de Dados Avançadas

Trabalho Prático

Aluno:

Tiago Oliveira 16622

Professor:

João Silva

1 de Abril de 2022

Conteúdo

1	Introdução	2
2	Estruturas de Dados	3
3	Propósitos e Objetivos	4
3.1	Definição de uma estrutura de dados dinâmica para a representação de um job com um conjunto finito de n operações;	4
3.2	Armazenamento/leitura de ficheiro de texto com representação de um job;	4
3.3	Inserção de uma nova operação;	6
3.4	Remoção de uma determinada operação;	7
3.5	Alteração de uma determinada operação;	8
3.6	Determinação da quantidade mínima de unidades de tempo necessárias para completar o job e listagem das respetivas operações;	8
3.7	Determinação da quantidade máxima de unidades de tempo necessárias para completar o job e listagem das respetivas operações;	9
3.8	Determinação da quantidade média de unidades de tempo necessárias para completar uma operação, considerando todas as alternativas possíveis;	10
4	Testes realizados	11
5	Conclusão	16

1 | Introdução

Requisitos do trabalho

Com este trabalho prático pretende-se sedimentar os conhecimentos relativos a definição e manipulação de estruturas de dados dinâmicas na linguagem de programação C. A essência deste trabalho reside no desenvolvimento de uma solução digital para o problema de escalonamento denominado Flexible Job Shop Problem (FJSSP). A solução a implementar 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).

Os materiais usados para a elaboração deste trabalho foram os seguintes componentes:



Figura 1.1: Computador



Figura 1.2: Git

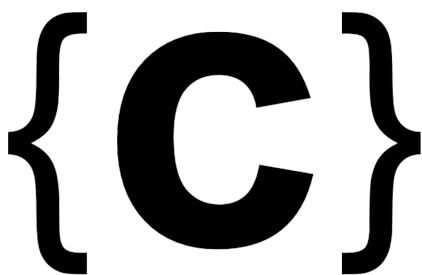


Figura 1.3: Linguagem C

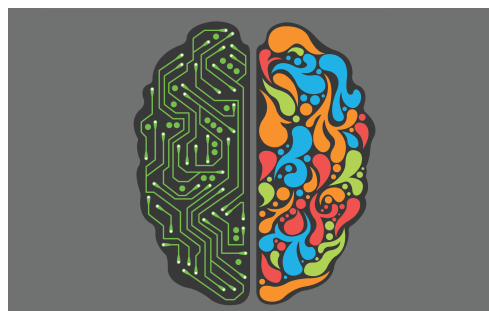


Figura 1.4: Logica

O Código está localizado na página de github: <https://github.com/TinoMeister/EDAProject>

2 | Estruturas de Dados

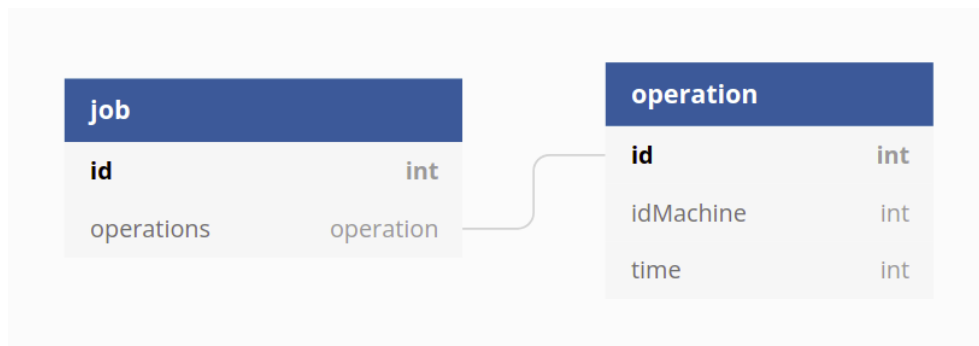


Figura 2.1: Estrutura

Para a realização deste projeto, idializou-se a seguinte estrutura:

1. Uma Operação contém um id, id para uma maquina e um tempo;
2. Um job contém um id e uma lista de operações;

Para esta fase inicial só se usou a tabela operation, mas mais a frente na fase 2 irá se usar a duas tabelas como se pode ver na figura acima 2.1.

3 | Propósitos e Objetivos

3.1 Definição de uma estrutura de dados dinâmica para a representação de um job com um conjunto finito de n operações;

Como já foi ilustrado na 2, podemos visualizar como foi feito a estrutura de dados para representar um job. Mas para melhor visualização de código a figura abaixo irá demonstrar como foi feito usando a linguagem C:

```
/**
 * @brief An struct type Operation.
 */
typedef struct oper
{
    int id;           /**< id of operation */
    int idMachine;    /**< id Machine */
    int time;         /**< time of the Machine */
    struct oper *next; /**< the next inserted*/
} Operation;
```

Figura 3.1: Estrutura em C

3.2 Armazenamento/leitura de ficheiro de texto com representação de um job;

Para armazenar e fazer a leituras num ficheiro de texto, definiu-se a seguinte forma de guardar os dados: *id;idMachine;time*, ou seja, guarda se o id da operação, o id da maquina e o seu tempo, usando o ; para os separar, como se pode ver na imagem a abaixo:

```
1;1;4
1;3;5
2;2;4
2;4;5
3;3;5
3;5;6
4;4;5
4;5;5
4;6;4
4;7;5
4;8;9
```

Figura 3.2:
Base de
Dados

```
// Save data of Object
void saveOpData(Operation* op)
{
    // Open file
    FILE *f = fopen("db/operations.txt", "w");

    // Verify if isnt null
    if (f == NULL)
    {
        printf("Error opening file!\n");
    }

    // Goes operation a operation and saves it into a line in the file
    while (op != NULL)
    {
        fprintf(f, "%d;%d;%d\n", op->id, op->idMachine, op->time);
        op = op->next;
    }

    // Close File
    fclose(f);
}
```

Para guardar no ficheiro, primeiro abre-se o ficheiro e enquanto a lista não for vazia iria guardar a informação de cada operação como ilustrado na imagem a cima 3.2 e pode ser ver o código ao lado:

Figura 3.3: Guardar no Ficheiro

Para ler no ficheiro, primeiro abre-se o ficheiro e lê-se linha a linha, após isso como ilustrado anteriormente 3.2, a informação está dividida por ;, portanto tem que se remover e por fim pode ser enviar por parametro a informação, no qual esta retorna uma Operação, como se poder ver na imagem ao lado:

```
// Load data of Operation
Operation* loadOpData(Operation* op)
{
    // Open file
    FILE *f = fopen("db/operations.txt", "r");
    char line[256];
    int id, idMachine, time;

    // Verify if isnt null
    if (f == NULL)
    {
        printf("Error opening file!\n");
        return NULL;
    }

    // Get the text line by line
    while (fgets(line, sizeof(line), f))
    {
        // Remove the ";" and get the values and insert into the list
        char *ptr = strtok(line, ";");
        id = (int)strtol(ptr, (char **)NULL, 10);

        ptr = strtok(NULL, ";");
        idMachine = (int)strtol(ptr, (char **)NULL, 10);

        ptr = strtok(NULL, ";");
        time = (int)strtol(ptr, (char **)NULL, 10);

        // Add the Operation
        op = addOperation(op, id, idMachine, time);
    }

    // Close File
    fclose(f);

    // Return the list
    return op;
}
```

Figura 3.4: Ler do Ficheiro

3.3 Inserção de uma nova operação;

Para a inserção de uma operação, passa-se por parametros a lista de tipo Operação e tres valores que são, o id, o id da maquina e o seu tempo. Após isso verifica se a operação é **NULL**, caso seja verdade, a nova informação é inserida e altera o sua variavel next para **NULL**, pois é o seu primerio registo.

Caso a lista nao seja nula, verifica se o id recebido é menor que o id da primeira posição da lista ou se o id for igual mas se o id da maquina for inferior do id da maquina da primeira posição da lista, caso seja verdade, então esta fica na posição inicial e altera o valor **.next** para o apontador da primeira posição da lista e altera a primeira posicao para a nova Operação inserida.

Caso não se verifique, por fim vai operação a operação na lista e verifica se o id recebido é maior que o id da próxima operação ou se os ids são iguais e os id da maquina recebido é maior que o id da maquina da proxima operação, se for verdade, então o valor **.next** da nova operação tera como apontador para a próxima operação e o operação atual o valor **.next** fica com da nova operação.

```
// Add new operation
Operation* addOperation(Operation* op, int id, int idMachine, int time)
{
    Operation *aux, *newOp = malloc(sizeof(Operation));
    bool exists = verifyOperation(op, id, idMachine);

    // Verify if can save a new operation into memory or alrely exists
    if (!newOp || exists) return NULL;

    // Save the information
    newOp->id = id;
    newOp->idMachine = idMachine;
    newOp->time = time;

    // Save in the first position if list is null
    if (!op)
    {
        newOp->next = NULL;
        op = newOp;
    }

    // Save in the first position if id is lower then the first id the list
    // or if id are equals but the idMachine is lower then the first idMachine
    else if (id < op->id || (id == op->id && idMachine < op->idMachine))
    {
        newOp->next = op;
        op = newOp;
    }
    else
    {
        aux = op;

        // While the id is higher then the id of element on the list gets the next element
        // or if the id is equal but the id machine is higher
        while (aux->next && (id > aux->next->id || (id == aux->next->id && idMachine > aux->next->idMachine))) aux = aux->next;

        // The new operation gets the pointer from the previous operation pointer
        newOp->next = aux->next;
        // The next operation gets the the pointer from the new operation
        aux->next = newOp;
    }

    // Return the operation list
    return op;
}
```

Figura 3.5: Adicionar Operação

3.4 Remoção de uma determinada operação;

Para a remoção de uma operação, passa-se por parametros a lista de operação e o seu index, caso o **index** seja 0, significa que esta-se perante a primeira posição, portanto a lista fica com a proxima posição e limpa se a anterior. Caso não esteja na primeira posição, faz se uma procura até achar a posição correta da Operação e após isso pega na Operação anterior e atualiza o apontador do valor **.next** para a proxima posição da Operação.

```
// Delete the operation
Operation* deleteOperation(Operation* op, int index)
{
    Operation *temp = op, *aux;
    int total = getSizeOp(op);

    // If the list is null or of the index is not between 1 and the total
    // of operations or if the operation already exists then return NULL
    if (!op || index <= 0 || index > total) return NULL;

    // If the index is 1 then gets the next element and put it as first and deletes the first
    if (index == 1)
    {
        op = temp->next;
        free(temp);
    }
    else
    {
        // Gets the previous element and the actual element
        for (int i = 1; i < index; i++)
        {
            aux = temp;
            temp = temp->next;
        }

        // If the actual element is NULL then return NULL
        if (!temp) return NULL;

        // Update the pointer so that the previous element goes to next element
        // of the actual element so it can be free
        aux->next = temp->next;

        free(temp);
    }

    return op;
}
```

Figura 3.6: Remover Operação

3.5 Alteração de uma determinada operação;

Para a alteração de uma operação, passa-se por parametros a lista, a posição onde se quer alterar a Operação e os novos valores. De seguida faz uma pesquisa até encontrar a localização da Operação pretendida e altera os valores antigos com os novos. Como se pode ver abaixo:

```
// Edit the operation
Operation* editOperation(Operation* op, int index, int id, int idMachine, int time)
{
    Operation *temp = op;
    int total = getSizeOp(op);
    bool exists = verifyOperation(op, id, idMachine);

    // If the list is null or of the index is not between 1 and the total
    // of operations or if the operation already exists then return NULL
    if (!op || index <= 0 && index > total || exists) return NULL;

    // Delete Operation
    temp = deleteOperation(temp, index);

    // If result is not NULL then add the Operation
    if (temp) temp = addOperation(temp, id, idMachine, time);

    // Return the new list of Operation
    return temp;
}
```

Figura 3.7: Alterar Operação

3.6 Determinação da quantidade mínima de unidades de tempo necessárias para completar o job e listagem das respetivas operações;

Para determinar a quantidade mínima de unidades de tempo, primeiro passa-se por parametros a lista e o id da operação, e cria-se uma variável do tipo **Operation** chamada lower e que vai a cada operação da lista e vai verificar se o id em que está é igual ao id se passou por parametro se for igual e se o for o menor tempo, então retorna essa operação.

```
// Show Operations with lower time
void showShorter(Operation* op)
{
    Operation *lower = malloc(sizeof(Operation));
    int totalOp = getTotalOp(op), count = 0;

    if (op != NULL)
    {
        printf("Id | IdMachine | Time\n");
        for (int i = 1; i <= totalOp; i++)
        {
            lower = getShorter(op, i);

            if (lower != NULL)
            {
                printf("%d | %d | %d\n", lower->id, lower->idMachine, lower->time);
                count += lower->time;
            }

            lower = NULL;
        }

        printf("Total: %d\n", count);
    }
}
```

Figura 3.8: Quantidade minima

```
// Get the lower timer to complete a operation
Operation* getShorter(Operation* op, int id)
{
    Operation *lower = NULL;

    // If the list is NULL return NULL
    if (!op)
        return NULL;

    while (op)
    {
        // Verify if the id is equal to the id of element in the list and if the
        // time NULL or if is higher then the time of the element if so the update
        if (id == op->id && (!lower || lower->time > op->time))
            lower = op;

        // Get the next element
        op = op->next;
    }

    // Return the operation with lowest time
    return lower;
}
```

Figura 3.9: Quantidade minima

3.7 Determinação da quantidade máxima de unidades de tempo necessárias para completar o job e listagem das respetivas operações;

Para determinar a quantidade máxima de unidades de tempo, a estrutura de resolução é muito semelhante com a do exercício explicado anteriormente 3.6, mas agora para ir buscar o maior tempo.

```
// Show Operations with higher time
void showLonger(Operation* op)
{
    Operation *higher = malloc(sizeof(Operation));
    int totalOp = getTotalOp(op), count = 0;

    if (op != NULL)
    {
        printf("Id | IdMachine | Time\n");
        for (int i = 1; i <= totalOp; i++)
        {
            higher = getLonger(op, i);

            if (higher != NULL)
            {
                printf("%d | %d | %d\n", higher->id, higher->idMachine, higher->time);
                count += higher->time;
            }

            higher = NULL;
        }

        printf("Total: %d\n", count);
    }
}
```

Figura 3.10: Quantidade maxima

```
// Get the higher timer to complete a operation
Operation* getLonger(Operation* op, int id)
{
    Operation *higher = NULL;

    // If the list is NULL return NULL
    if (!op)
        return NULL;

    while (op)
    {
        // Verify if the id is equal to the id of element in the list and if the
        // time NULL or if is lower then the time of the element if so the update
        if (id == op->id && (!higher || higher->time < op->time))
            higher = op;

        // Get the next element
        op = op->next;
    }

    // Return the operation with highest time
    return higher;
}
```

Figura 3.11: Quantidade maxima

3.8 Determinação da quantidade média de unidades de tempo necessárias para completar uma operação, considerando todas as alternativas possíveis;

Para determinar a quantidade média de unidades de tempo, inicialmente começa-se por fazer leitura na lista toda e verificando se existe alguma Operação com id igual ao que se está a procurar, caso haja, soma-se o tempo dessa Operação e acrescenta-se mais um a uma variável **count** que quando acabar a lista toda essa soma de de tempo se'lla divida pela variável **count** no qual dará a média de tempo para cara Operação.

```
// Show Operations average time
void showAverage(Operation* op)
{
    int totalOp = getTotalOp(op);
    float result = 0, count = 0;

    if (op != NULL)
    {
        printf("Id | Time\n");
        for (int i = 1; i <= totalOp; i++)
        {
            result = getAverage(op, i);

            if (result > -1)
            {
                printf("%d | %.2f\n", i, result);
                count += result;
            }
        }

        printf("Total: %.2f\n", count);
    }
}
```

Figura 3.12: Media

```
// Get the average timer to complete a operation
float getAverage(Operation* op, int id)
{
    int count = 0, total = 0;

    while (op)
    {
        // If the id is equal to the id of the element on the list then increase plus
        // one to the count and increase the time value
        if (id == op->id)
        {
            total += op->time;
            count++;
        }

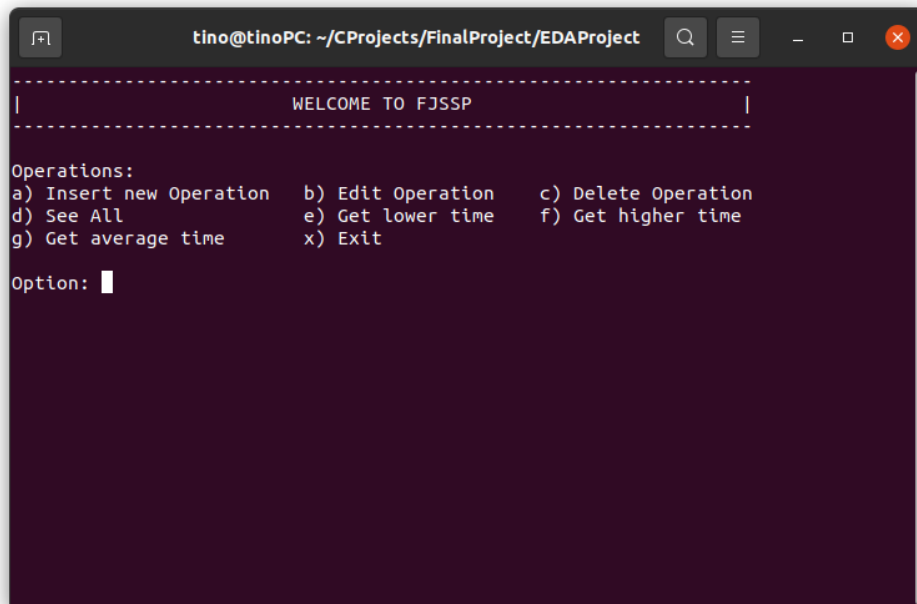
        // Get the next element
        op = op->next;
    }

    // If count equals to 0 then return -1 else return the total of time divided by
    // total of operations
    if (count == 0)
        return -1;
    else
        return total / count;
}
```

Figura 3.13: Media

4 | Testes realizados

Página Principal



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject

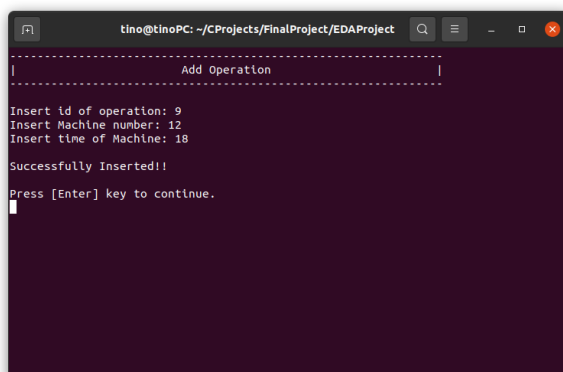
|----- WELCOME TO FJSSP -----|

Operations:
a) Insert new Operation    b) Edit Operation    c) Delete Operation
d) See All                e) Get lower time    f) Get higher time
g) Get average time       x) Exit

Option: 
```

Figura 4.1: Página Principal

Exemplo de Adicionar uma Operação



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject

|----- Add Operation -----|

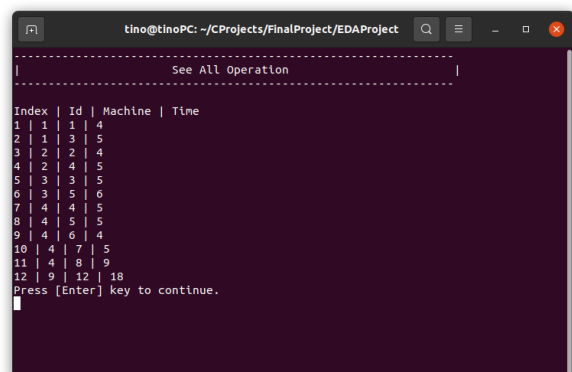
Insert id of operation: 9
Insert Machine number: 12
Insert time of Machine: 18

Successfully Inserted!!

Press [Enter] key to continue.

```

Figura 4.2: Adicionar



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject

|----- See All Operation -----|

Index | Id | Machine | Time
1 | 1 | 1 | 4
2 | 1 | 3 | 5
3 | 2 | 2 | 4
4 | 2 | 4 | 5
5 | 3 | 3 | 5
6 | 3 | 5 | 6
7 | 4 | 4 | 5
8 | 4 | 5 | 5
9 | 4 | 6 | 4
10 | 4 | 7 | 5
11 | 4 | 8 | 9
12 | 9 | 12 | 18

Press [Enter] key to continue.

```

Figura 4.3: Verificar

Exemplo de Editar uma Operação

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|----- Edit Operation -----|
Index | Id | Machine | Time
1 | 1 | 1 | 4
2 | 1 | 3 | 5
3 | 2 | 2 | 4
4 | 2 | 4 | 5
5 | 3 | 3 | 5
6 | 3 | 5 | 6
7 | 4 | 4 | 5
8 | 4 | 5 | 5
9 | 4 | 6 | 4
10 | 4 | 7 | 5
11 | 4 | 8 | 9
12 | 9 | 12 | 18

Insert index to update: 12
Insert id of operation to change: 7
Insert Machine number to change: 3
Insert time of Machine to change: 30
Successfully Updated!!

Press [Enter] key to continue.
```

Figura 4.4: Editar

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|----- See All Operation -----|
Index | Id | Machine | Time
1 | 1 | 1 | 4
2 | 1 | 3 | 5
3 | 2 | 2 | 4
4 | 2 | 4 | 5
5 | 3 | 3 | 5
6 | 3 | 5 | 6
7 | 4 | 4 | 5
8 | 4 | 5 | 5
9 | 4 | 6 | 4
10 | 4 | 7 | 5
11 | 4 | 8 | 9
12 | 7 | 3 | 30
Press [Enter] key to continue.
```

Figura 4.5: Verificar

Exemplo de Remover uma Operação

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|----- Delete Operation -----|
Index | Id | Machine | Time
1 | 1 | 1 | 4
2 | 1 | 3 | 5
3 | 2 | 2 | 4
4 | 2 | 4 | 5
5 | 3 | 3 | 5
6 | 3 | 5 | 6
7 | 4 | 4 | 5
8 | 4 | 5 | 5
9 | 4 | 6 | 4
10 | 4 | 7 | 5
11 | 4 | 8 | 9
12 | 7 | 3 | 30

Insert index to delete: 12
Successfully Deleted!!

Press [Enter] key to continue.
```

Figura 4.6: Remover

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|----- See All Operation -----|
Index | Id | Machine | Time
1 | 1 | 1 | 4
2 | 1 | 3 | 5
3 | 2 | 2 | 4
4 | 2 | 4 | 5
5 | 3 | 3 | 5
6 | 3 | 5 | 6
7 | 4 | 4 | 5
8 | 4 | 5 | 5
9 | 4 | 6 | 4
10 | 4 | 7 | 5
11 | 4 | 8 | 9
Press [Enter] key to continue.
```

Figura 4.7: Verificar

Exemplo de Ver todas as Operações

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|----- See All Operation -----|
Index | Id | Machine | Time
1 | 1 | 1 | 4
2 | 1 | 3 | 5
3 | 2 | 2 | 4
4 | 2 | 4 | 5
5 | 3 | 3 | 5
6 | 3 | 5 | 6
7 | 4 | 4 | 5
8 | 4 | 5 | 5
9 | 4 | 6 | 4
10 | 4 | 7 | 5
11 | 4 | 8 | 9
12 | 9 | 12 | 14
13 | 9 | 14 | 20
Press [Enter] key to continue.
```

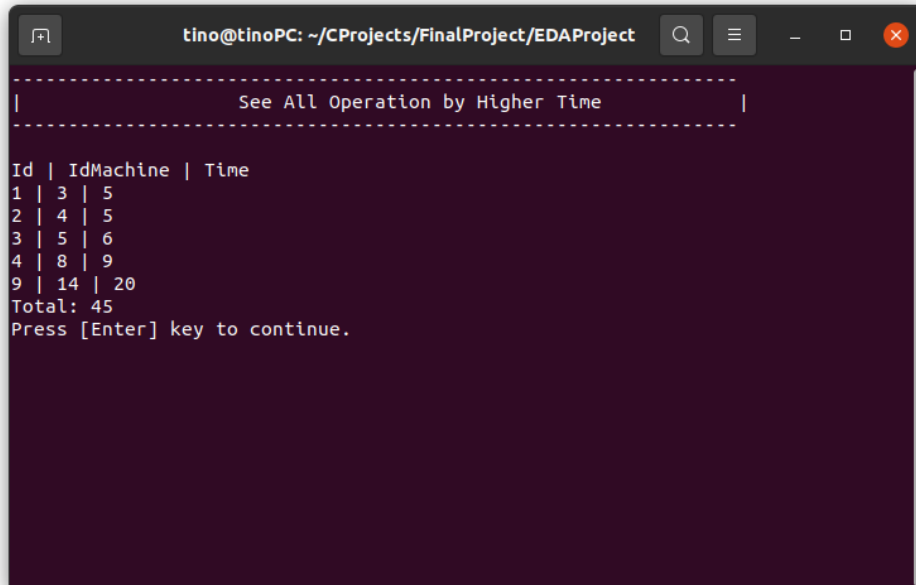
Figura 4.8: Ver todas

Exemplo de ver o menor tempo por cada Operação

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|----- See All Operation by Lower Time -----|
Id | IdMachine | Time
1 | 1 | 4
2 | 2 | 4
3 | 3 | 5
4 | 6 | 4
9 | 12 | 14
Total: 31
Press [Enter] key to continue.
```

Figura 4.9: Menor tempo

Exemplo de ver o maior tempo por cada Operação

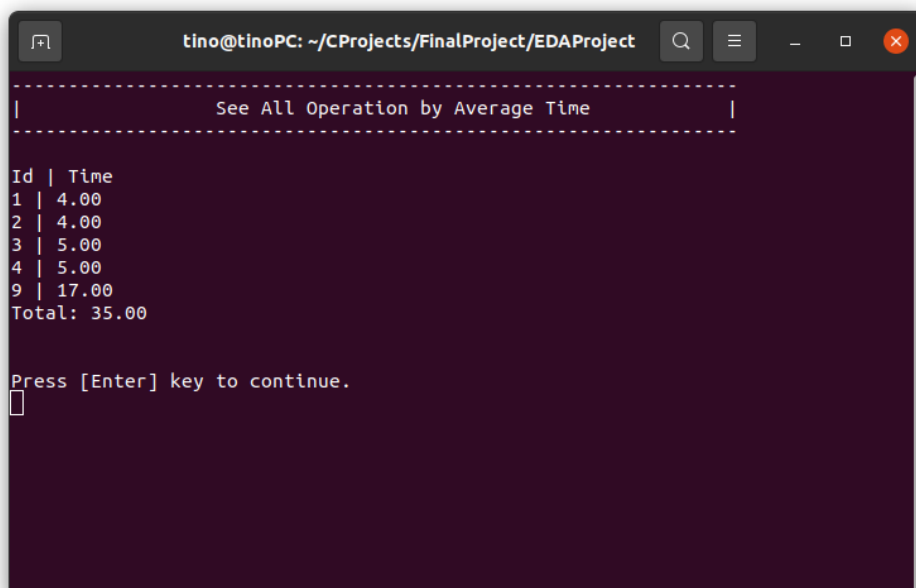


```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|-----|
|               See All Operation by Higher Time               |
|-----|

Id | IdMachine | Time
1  | 3         | 5
2  | 4         | 5
3  | 5         | 6
4  | 8         | 9
9  | 14        | 20
Total: 45
Press [Enter] key to continue.
```

Figura 4.10: Maior tempo

Exemplo de ver o média tempo por cada Operação



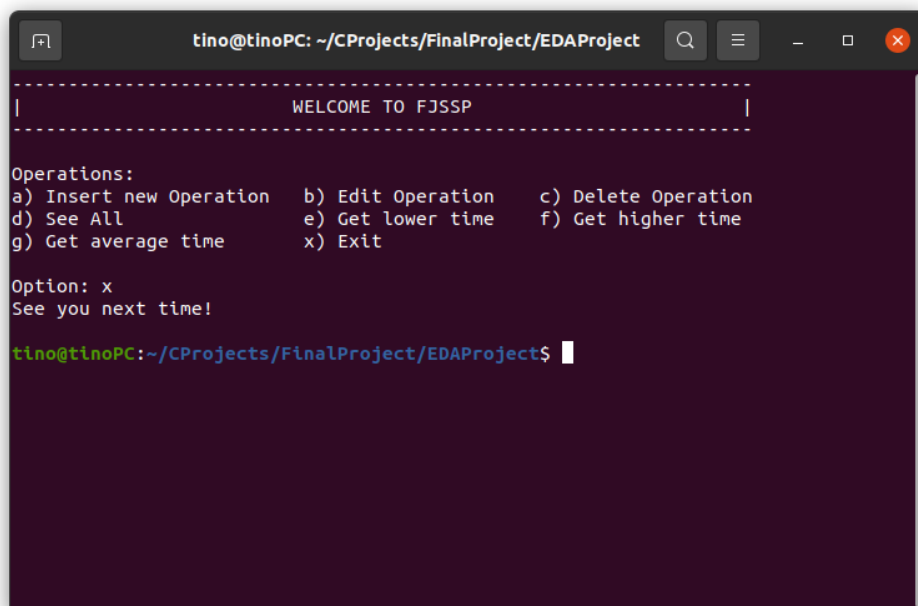
```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
|-----|
|               See All Operation by Average Time               |
|-----|

Id | Time
1  | 4.00
2  | 4.00
3  | 5.00
4  | 5.00
9  | 17.00
Total: 35.00

Press [Enter] key to continue.
█
```

Figura 4.11: Media

Exemplo de sair do Programa



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProject
-----
|                                     WELCOME TO FJSSP                                     |
-----

Operations:
a) Insert new Operation    b) Edit Operation    c) Delete Operation
d) See All                e) Get lower time    f) Get higher time
g) Get average time       x) Exit

Option: x
See you next time!

tino@tinoPC:~/CProjects/FinalProject/EDAProject$
```

Figura 4.12: Sair

5 | Conclusão

Em suma, o projeto apresentado corresponde com o professor requesitou, todo o código apresentado foi feito pensando na melhor estratégia de como resolver o problema, sabendo que na área de programação nenhum código é 100% correto nem errado, mas sim resolvido de uma forma diferente. Contudo pode se afirmar que a primeira fase está concluída.