



**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 representação de um conjunto finito de m jobs associando a cada job um determinado conjunto finito de operações;	4
3.2	Armazenamento/leitura de ficheiro de texto com representação de um process plan; . .	4
3.3	Inserção de um novo job;	6
3.4	Remoção de um job;	7
3.5	Inserção de uma nova operação num job;	8
3.6	Remoção de uma determinada operação de um job;	9
3.7	Edição das operações associadas a um job;	11
3.8	Cálculo de uma proposta de escalonamento para o problema FJSSP;	12
3.9	Representação de diferentes process plan;	19
4	Testes realizados	20
5	Shifting Bottleneck	25
6	Conclusão	28

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

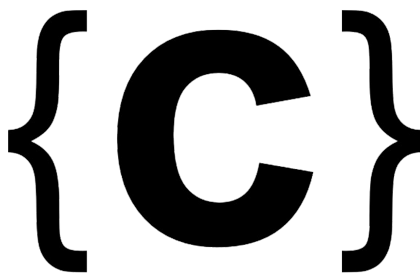


Figura 1.3: Linguagem C



Figura 1.4: Logica

O Código está localizado na página de github: <https://github.com/TinoMeister/EDAProject-V2>

2 | Estruturas de Dados

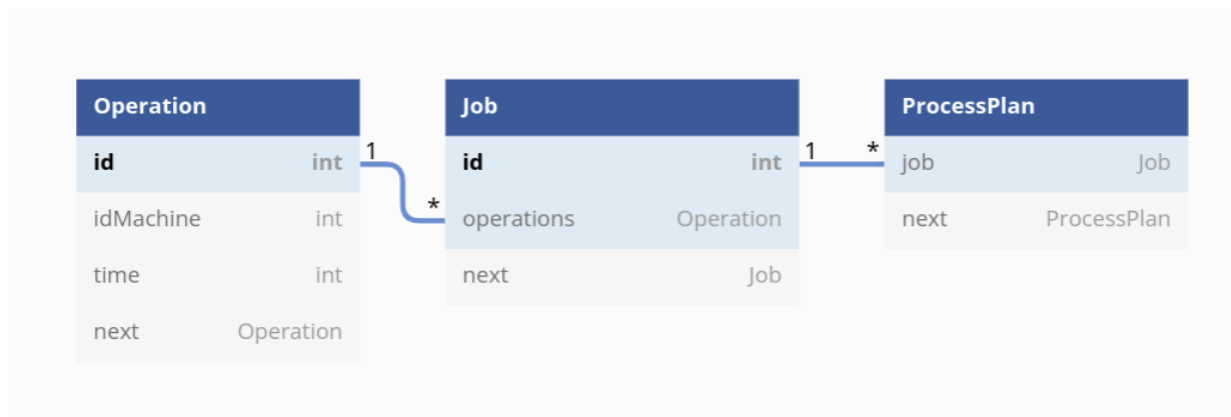


Figura 2.1: Estrutura

Para a realização deste projeto, idealizou-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;
3. Um Process plan contém uma lista de jobs;

3 | Propósitos e Objetivos

3.1 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;

Como já foi ilustrado na 2, podemos visualizar como foi feito a estrutura de dados para representar uma lista de jobs. 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

```
/**
 * @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 3.2: Estrutura em C

```
/**
 * @brief A struct of Process Plan.
 * A list of all the jobs.
 */
typedef struct pPlan
{
    struct job *jb;
    struct pPlan *next;
} ProcessPlan;
```

Figura 3.3: Estrutura em C

3.2 Armazenamento/leitura de ficheiro de texto com representação de um process plan;

Para armazenar e fazer a leituras num ficheiro de texto, defeniu-se a seguinte forma de guardar os dados: *idJob;idOperation;idMachine;time*, ou seja, guarda se o id do job, 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;1;4
1;1;3;5
1;2;2;4
1;2;4;5
1;3;3;5
1;3;5;6
1;4;4;5
1;4;5;5
1;4;6;4
1;4;7;5
1;4;8;9
```

Figura 3.4:
Base de
Dados

```
// Save data of Object
void saveData(Job* jb)
{
    Operation *temp;

    // Open file
    FILE *f = fopen("db/db.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
    if (jb != NULL)
    {
        while (jb != NULL)
        {
            if (jb->op != NULL)
            {
                temp = jb->op;
                while (temp != NULL)
                {
                    fprintf(f, "%d;%d;%d\n", jb->id, temp->id, temp->idMachine, temp->time);
                    temp = temp->next;
                }
            }
            jb = jb->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 3.4 e pode ser ver o código ao lado:

Figura 3.5: Guardar no Ficheiro

Para ler no ficheiro, primeiro abre-se o ficheiro e lê-se linha a linha, após isso como ilustrado anteriormente 3.4, 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 lista de Jobs, como se poder ver na imagem ao lado:

```
// Load data of Jobs
Job* loadData(Job* jb)
{
    // Open file
    FILE *f = fopen("db/db.txt", "r");
    char line[256];
    int idJob, idOp, idMachine, time, count = 0;

    // 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, ";");
        idJob = (int)strtol(ptr, (char **)NULL, 10);

        ptr = strtok(NULL, ";");
        idOp = (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);

        //printf("%d %d %d %d\n", idJob, idOp, idMachine, time);

        if (idJob != count)
        {
            jb = addJob(jb, idJob);
            count = idJob;
        }

        jb = addOpJob(jb, idJob, idOp, idMachine, time);
    }

    // Close File
    fclose(f);

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

Figura 3.6: Ler do Ficheiro

3.3 Inserção de um novo job;

Para inserir um novo job, passa se por paramentro a lista de jobs, e o id novo do job. É verificado se ao alocar um novo Job é possível e se o id é superior a 0. Após isso guarda se o id no novo job e este é adicionado no fim da lista, finalizando, esta lista é retornada.

```
Job* addJob(Job* jb, int id)
{
    Job *aux, *newJob = malloc(sizeof(Job));

    // Verify if can save a new job into memory
    if (!newJob || id <= 0) return NULL;

    // Save the information
    newJob->id = id;
    newJob->op = NULL;

    // Save in the first position if list is null
    if (!jb)
    {
        newJob->next = NULL;
        jb = newJob;
    }
    else if (id < jb->id) // Save in the first position if id is lower then the first id of the list
    {
        newJob->next = jb;
        jb = newJob;
    }
    else
    {
        aux = jb;

        // While the id is higher then the id of element on the list gets the next element
        while (aux->next && id > aux->next->id) aux = aux->next;

        // Save the new job between the elements
        newJob->next = aux->next;
        aux->next = newJob;
    }

    // Return the job list
    return jb;
}
```

Figura 3.7: Adicionar Job

3.4 Remoção de um job;

Para remover um job, passa se por paramentro a lista de jobs, e o index da localização do job. É verificado se a lista Job é **NULL** e se index é superior a 0 ou se é superior ao total de jobs existentes na lista. Após essa verificação, esse job é removido e a lista é retornada.

```
Job* deleteJob(Job* jb, int index)
{
    Job *temp = jb, *aux;
    int total = getSizeJb(jb);

    // If the list is null or of the index is not bettween 1 and the total of job then return NULL
    if (!jb || 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)
    {
        jb = 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 list
    return jb;
}
```

Figura 3.8: Remover Job

3.5 Inserção de uma nova operação num job;

Para adicionar uma operação ao Job, passa-se por parametro a lista de jobs, o id do job, o id da operação, o id da máquina e o seu tempo, existe uma verificação para estes parametros não serem nulos nem menores que 0 e após isso vai-se buscar o elemento que contém o id job recebido e adiciona-se a Operação, usando a função explicada no relatório anterior e que estará presente na linha abaixo.

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 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 primeiro registo.

Caso a lista não seja nula, verifica-se 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 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 terá como apontador para a próxima operação e o operação atual o valor **.next** fica com da nova operação.

```
Job* addOpJob(Job* jb, int idJob, int idOp, int idMachine, int time)
{
    Job *temp = jb;
    Operation *result = NULL;

    // If job is NULL return NULL
    if (!jb || idOp <= 0 || idMachine <= 0 || time <= 0) return NULL;

    // Gets the next element until the id of the list is different of the id of the job received
    while (temp && temp->id != idJob) temp = temp->next;

    // If the element is not NULL then add a new Operation to that Job
    if (temp) result = addOperation(temp->op, idOp, idMachine, time);

    // If result is NULL then something went wrong and return NULL
    if (result && result->id == -1) return NULL;

    // Update the list of Operations
    temp->op = result;

    // Return job list
    return jb;
}
```

Figura 3.9: Adicionar Operação a Job

```
// 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 already 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 3.10: Adicionar Operação

3.6 Remoção de uma determinada operação de um job;

Para remover uma operação ao Job, passa se por parametro a lista de jobs, o id do job e o index da operação, existe uma verificação para estes parametros não serem nulos nem menores que 0 e após isso vai se buscar o elemento que contem o id job recebido e remove se a Operação, usando a função explicada no relatório anterior e que estará presente na linha abaixo.

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.

```
Job* deleteOpJob(Job* jb, int idJob, int indexOp)
{
    Job *temp = jb;
    Operation *result = NULL;

    // If job is NULL return NULL
    if (!jb || idJob <= 0 || indexOp <= 0) return NULL;

    // Gets the next element until the id of the list is different of the id of the job received
    while (temp && temp->id != idJob) temp = temp->next;

    // If the element is not NULL then delete the Operation to that Job
    if (temp) result = deleteOperation(temp->op, indexOp);

    // If result is NULL then something went wrong and return NULL
    if (result && result->id == -1) return NULL;

    // Update the list of Operations
    temp->op = result;

    // Return job list
    return jb;
}
```

Figura 3.11: Remover Operação a Job

```
// 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 3.12: Remover Operação

3.7 Edição das operações associadas a um job;

Para editar uma operação ao Job, passa se por parametro a lista de jobs, o id do job, o index da operação, o id da operação, o id da máquina e o seu tempo, existe uma verificação para estes parametros não serem nulos nem menores que 0 e após isso vai se buscar o elemento que contem o id job recebido e adiciona se a Operação, usando a função explicada no relatório anterior e que estará presente na linha abaixo.

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.

```
Job* editOpJob(Job* jb, int idJob, int indexOp, int idOp, int idMachine, int time)
{
    Job *temp = jb;
    Operation *result = NULL;

    // If job is NULL return NULL
    if (!jb || idJob <= 0 || indexOp <= 0 || idOp <= 0 || idMachine <= 0 || time <= 0) return NULL;

    // Gets the next element until the id of the list is diferent of the id of the job received
    while (temp && temp->id != idJob) temp = temp->next;

    // If the element if not NULL then edit the Operation to that Job
    if (temp) result = editOperation(temp->op, indexOp, idOp, idMachine, time);

    // If result is NULL then something went wrong and return NULL
    if (result && result->id == -1) return NULL;

    // Update the list of Operations
    temp->op = result;

    // Return job list
    return jb;
}
```

Figura 3.13: Editar Operação a Job

```
// Edit the operation
Operation* editOperation(Operation* op, int index, int id, int idMachine, int time)
{
    Operation *temp = op;
    int total = getSizeOp(op), count = 1;

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

    // Get the operation representing the index
    while (temp && count < index)
    {
        count++;
        temp = temp->next;
    }

    // If the id and id Machine are still equal to the element then just update the time
    if (temp->id == id && temp->idMachine == idMachine) temp->time = time;
    else
    {
        // Add the Operation
        temp = addOperation(op, id, idMachine, time);

        // If result is not NULL then delete the Operation
        if (temp) temp = deleteOperation(op, index);
    }

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

Figura 3.14: Editar Operação

3.8 Cálculo de uma proposta de escalonamento para o problema FJSSP;

Para realizar o escalonamento do problema, foi usada uma técnica chamada *Shifting Bottleneck*, que retorna as melhores opções de escalonar um conjunto de jobs por uma máquina. Para perceber melhor esta técnica, estará explicado na parte *Shifting Bottleneck*.

Para realizar este processo, criou se mais 2 estruturas para guardar esta informação e ajudar no escalonamento.

Estrutura *Shifting Bottleneck*, guarda os dados explicados na parte *Shifting Bottleneck*.

Estrutura *Ordens*, onde guarda um array de listas *Shifting Bottleneck*, o tempo total, o lowerBone e o tamanho da lista.

```
typedef struct shiftingBottleneck
{
    int idMachine;
    int idJb;
    int idOp;
    int time;
    int waitTime;
    int timeFinish;
    struct shiftingBottleneck *next;
} ShiftingBottleneck;
```

Figura 3.15: Estrutura Shifting Bottleneck

```
typedef struct orders
{
    struct shiftingBottleneck **data;
    int time;
    int lowerBone;
    int size;
    struct orders *next;
} Orders;
```

Figura 3.16: Estrutura Ordens

```
printf("-----\n");
printf("|          ESCALATION          |\n");
printf("-----\n\n");

start = clock();

finalJb = getCombinations(jb);
combJb = getBestCombinations(finalJb);
finalPPlan = getCombinationsJb(combJb);
getPossibilities(finalPPlan);

end = clock();
time_taken = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("fun() took %f seconds to execute \n", time_taken);
printf("File saved at db/bestEscalation.txt\n\n");
break;
```

Figura 3.17: Funções

Primeiro vai se buscar todas as combinações de operações que cada Job têm.

```
Job* getCombinations(Job* jb)
{
    Job *combJb = NULL, *resultJob = NULL;
    Operation *op1, *op2, *tempOp;
    int totalJob;

    // If list is NULL, then return NULL
    if (!jb) return NULL;

    // Goes Job by Job
    while (jb)
    {
        op1 = jb->op;

        // Goes Operation by Operation while th id of Operation is equals to 1
        while (op1 && op1->id == 1)
        {
            // Creates a Process Plan with the id of the job and the operation
            combJb = createJob(combJb, jb->id, op1);

            // Gets the next element and puts it into the variable op2
            op2 = op1->next;

            // Goes through the whole list
            while (op2)
            {
                // If the Ids are diferent, then continous
                if (op2->id > op1->id)
                {
                    // Get the pointer of the first element on the list thats doesn't have the Operation 2
                    resultJob = getJobOp(combJb, jb->id, op1, op2);
                    // Get the total number of Process Plan
                    totalJob = getTotalJb(resultJob);

                    // Goes through the whole list get in the previous step
                    for (int i = 1; i <= totalJob; i++)
                    {
                        tempOp = resultJob->op;

                        // Verify if exists doesn't have Operation 2
                        if (!existsOp(tempOp, op2))
                        {
                            // Creates a Process Plan with the id of the job and the operation
                            combJb = createJobOp(combJb, jb->id, tempOp);

                            // All the Operation 2 into the list as last element
                            combJb = addJobOp(combJb, jb->id, op2);
                        }

                        resultJob = resultJob->next;
                    }
                }
                op2 = op2->next;
            }
            op1 = op1->next;
        }
    }
}
```

Figura 3.18: Combinações por Operação

Segundo seria ir se buscar todas as combinações que cada Job pode formar, contando com as combinações de Operações, mas fazendo, faria com que o tempo de processamento seria elevado, portanto para este exemplo foi feito uma função que vai buscar os Jobs que aparecem primeiro que contem todas as maquinas, e que o total de jobs por maquina não ultrapasse 5.

```
Job* getBestCombinations(Job* jb)
{
    Job *combJb = NULL, *temp, *ant;
    Operation *tempOp;
    int time, idJob = 0, average = 0, higher = 0;

    if (!jb) return NULL;

    while (jb)
    {
        if (jb->id != idJob)
        {
            //idJob = jb->id;
            combJb = createJobOp(combJb, jb->id, jb->op);

            if (verifyIsComp(combJb)) idJob = jb->id;
            else
            {
                temp = combJb;
                while (temp->next)
                {
                    ant = temp;
                    temp = temp->next;
                }

                ant->next = NULL;
                free(temp);
            }
        }

        jb = jb->next;
    }

    return combJb;
}
```

Figura 3.19: Buscar as melhores combinações por Operação

Terceiro, vai se buscar todas as combinações que cada Job pode formar, contando com as combinações de Operações, pois agora ja é possível porque ja foi reduzido o total de combinações de Operações.

```
ProcessPlan* getCombinationsJb(Job* jb)
{
    ProcessPlan *pplan = NULL, *resultPPlan = NULL;
    Job *temp1 = jb, *temp2, *tempJb;
    Operation *tempOp;
    int totalPPlan;

    // Goes job a job until the job id is different then 1
    while (temp1 && temp1->id == 1)
    {
        // Create a new Process Plan with the temp1
        pplan = createPPlanOp(pplan, temp1);

        // Get the next job and set into temp2
        temp2 = temp1->next;

        while (temp2)
        {
            // Verify if the id of temp1 is higher then temp1
            if (temp2->id > temp1->id)
            {
                // Get the Process Plan and get the total of process plan that exists
                resultPPlan = getPPlanJb(pplan, temp1, temp2);
                totalPPlan = getTotalPPlan(resultPPlan);

                for (int i = 1; i <= totalPPlan; i++)
                {
                    tempJb = resultPPlan->jb;

                    // Verify if exists temp2 in inside on tempJb
                    if (!existsJb(tempJb, temp2))
                    {
                        // If not, then create a new Process Plan whit tempJb and add the new job, temp2
                        pplan = createPPlanJb(pplan, tempJb);
                        pplan = addPPlanJb(pplan, temp2);
                    }

                    resultPPlan = resultPPlan->next;
                }
            }
            temp2 = temp2->next;
        }
        temp1 = temp1->next;
    }

    // Remove the elements that are incompleted
    pplan = removeGarbagedJb(pplan);

    return pplan;
}
```

Figura 3.20: Combinações por Job

Por fim, agora sim já se pode fazer o escalonamento e este processo é muito complexo, pois como já referido anteriormente, este usa o método *Shifting Bottleneck*.

```
void getPossibilities(ProcessPlan* pplan)
{
    Job *jb;
    ShiftingBottleneck *data = NULL;
    Orders *bestOrders;
    TopOrders *topOrders = NULL;

    int *lowerBoneM, *done;
    int totalMachine, totalJb, timeSpan = 0;

    if (!pplan) return;

    // Get total jobs and machines
    totalJb = getTotalJb(pplan->jb);
    totalMachine = getTotalMachine(pplan);

    // Allocate in memory a array for save all jobs that has been done
    done = malloc(totalMachine * sizeof(int));

    // goes pplan to pplan
    while (pplan)
    {
        // Gets the Job
        jb = pplan->jb;

        for (int i = 0; i < totalMachine; i++)
        {
            // Get lower Bone
            lowerBoneM = getLowerBoneM(jb, totalMachine, done);
            done[i] = lowerBoneM[0];
            if (timeSpan < lowerBoneM[1]) timeSpan = lowerBoneM[1];

            // Get all the Jobs that has the machine
            data = getData(jb, lowerBoneM[0], timeSpan);

            // Get best order for escalation
            bestOrders = getBestOrder(data, lowerBoneM[0], totalJb);
            timeSpan += bestOrders->lowerBone;

            // Save order
            topOrders = saveTopOrders(topOrders, bestOrders, jb, i, totalMachine, timeSpan);
        }

        pplan = pplan->next;
    }

    // Get best escalation
    getBestEscalation(topOrders);
}
```

Figura 3.21: Escalonamento Parte 1

Primeiro vai se buscar os Jobs em que estão associados ao id da maquina recebido e calcula se o seu tempo, o seu tempo de espera e o seu tempo de acabar.

```

ShiftingBottleneck* getDataJob* jb, int idMachine, int maxTime)
{
    ShiftingBottleneck* data, *list = NULL, *temp;
    Operation* op;
    int time, waitTime, timeFinish, totalMac, count, idOp;

    if (!jb) return NULL;

    while (jb)
    {
        // Gets the total machine that exists inside of Operation by idMachine
        totalMac = getTotalMachByIDMac(jb->op, idMachine);

        for (int i = 0; i < totalMac; i++)
        {
            op = jb->op;
            time = 0;
            waitTime = 0;
            timeFinish = 0;
            count = 0;
            data = malloc(sizeof(ShiftingBottleneck));

            while (op)
            {
                // Verify of the id Machine and count is equal to the i and save the
                if (op->idMachine == idMachine && count == i)
                {
                    idOp = op->id;
                    time = op->time;
                    count++;
                }
                else if (time == 0) waitTime += op->time; // If time is 0 then encrea
                else if (time != 0) timeFinish += op->time; // If time is 0 then enc
                // If the id machine is equal to id Machine and the count is different
                if (op->idMachine == idMachine && count != i) count++;

                op = op->next;
            }

            // If time is different != 0, then save the data
            if (time != 0)
            {
                data->idMachine = idMachine;
                data->idOp = idOp;
                data->time = time;
                data->waitTime = waitTime;
                data->timeFinish = maxTime - timeFinish;
                data->next = NULL;

                if (!list) list = data;
                else
                {
                    temp = list;
                    while (temp->next) temp = temp->next;
                    temp->next = data;
                }
            }

            jb = jb->next;
        }

        // return the list of ShiftingBottleneck
        return list;
    }
}

```

Figura 3.22: Escalonamento Parte 2

Depois, usa-se o método *Shifting Bottleneck*, para buscar a melhor posição que cada Job tem que estar para se pode escalonar.

```

ShiftingBottleneck* getDataJob* jb, int idMachine, int maxTime)
{
    ShiftingBottleneck* data, *list = NULL, *temp;
    Operation* op;
    int time, waitTime, timeFinish, totalMac, count, idOp;

    if (!jb) return NULL;

    while (jb)
    {
        // Gets the total machine that exists inside of Operation by idMachine
        totalMac = getTotalMachByIDMac(jb->op, idMachine);

        for (int i = 0; i < totalMac; i++)
        {
            op = jb->op;
            time = 0;
            waitTime = 0;
            timeFinish = 0;
            count = 0;
            data = malloc(sizeof(ShiftingBottleneck));

            while (op)
            {
                // Verify of the id Machine and count is equal to the i and save the
                if (op->idMachine == idMachine && count == i)
                {
                    idOp = op->id;
                    time = op->time;
                    count++;
                }
                else if (time == 0) waitTime += op->time; // If time is 0 then encrea
                else if (time != 0) timeFinish += op->time; // If time is 0 then enc
                // If the id machine is equal to id Machine and the count is different
                if (op->idMachine == idMachine && count != i) count++;

                op = op->next;
            }

            // If time is different != 0, then save the data
            if (time != 0)
            {
                data->idMachine = idMachine;
                data->idOp = idOp;
                data->time = time;
                data->waitTime = waitTime;
                data->timeFinish = maxTime - timeFinish;
                data->next = NULL;

                if (!list) list = data;
                else
                {
                    temp = list;
                    while (temp->next) temp = temp->next;
                    temp->next = data;
                }
            }

            jb = jb->next;
        }

        // return the list of ShiftingBottleneck
        return list;
    }
}

```

Figura 3.23: Escalonamento Parte 2

```

shiftingBottleneck.getData() jb, int idMachine, int maxTime)
{
    ShiftingBottleneck *data, *list = NULL, *temp;
    Operation *op;
    int time, waitTime, timeFinish, totalMac, count, idOp;

    if (!jb) return NULL;

    while (jb)
    {
        // Gets the total machine that exists inside of Operation by idMachine
        totalMac = getTotalMachineByIdOp(jb->op, idMachine);

        for (int i = 0; i < totalMac; i++)
        {
            op = jb->op;
            time = 0;
            waitTime = 0;
            timeFinish = 0;
            count = 0;
            data = malloc(sizeof(ShiftingBottleneck));

            while (op)
            {
                // Verify of the id Machine and count is equal to the i and save the
                if (op->idMachine == idMachine && count == i)
                {
                    idOp = op->id;
                    time = op->time;
                    count++;

                }
                else if (time == 0) waitTime += op->time; // If time is 0 then encrea
                else if (time != 0) timeFinish = op->time; // If time is 10 then enc
                // If the id machine is equal to id Machine and the count is different
                if (op->idMachine == idMachine && count != i) count++;

            }
            op = op->next;

            // If time is different != 0, then save the data
            if (time != 0)
            {
                data->idMachine = idMachine;
                data->idOp = jb->id;
                data->time = idOp;
                data->waitTime = time;
                data->timeFinish = maxTime - timeFinish;
                data->next = NULL;

            }
            if (!list) list = data;
            else
            {
                temp = list;
                while (temp->next) temp = temp->next;
                temp->next = data;
            }

            jb = jb->next;

        }

        // return the list of ShiftingBottleneck
        return list;
    }
}

```

Figura 3.24: Escalonamento Parte 3

Por fim guarda-se a informação pronta para demonstrar.

```

void getBestEscalation(TopOrders* topOrders)
{
    TopOrders* finalOrders = NULL;
    Job *tempJob;
    Orders *tempOr;
    int totalOp, totalMac;

    tempJob = topOrders->jb;
    totalOp = getTotalOpJob(tempJob);
    totalMac = getTotalMachineJob(tempJob);

    // Save element in order by machine
    for (int i = 1; i <= totalOp; i++)
    {
        for (int j = 0; j < totalMac; j++)
        {
            tempOr = topOrders->orders[j];

            while (tempOr)
            {
                for (int k = 0; k < tempOr->size && tempOr->data[k]; k++)
                {
                    if (tempOr->data[k]->idOp == i)
                    {
                        finalOrders = saveFinalOrders(finalOrders, topOrders, tempOr, j, k);
                    }
                }

                tempOr = tempOr->next;
            }
        }
    }

    // Save escalation to file
    saveEscalation(finalOrders, totalMac);
}

```

Figura 3.25: Escalonamento Parte 4

3.9 Representação de diferentes process plan;

Para resolver este problema, usou se o método do problema anterior, mas só se pede a localização do ficheiro que se quer escalonar.

```
printf("-----\n");
printf("|          DIFFERENT ESCALATION          |\n");
printf("-----\n\n");

printf("Insert location of file: ");
scanf("%s", locationFile);

jbTemp = loadDataByLoc(jbTemp, locationFile);

start = clock();

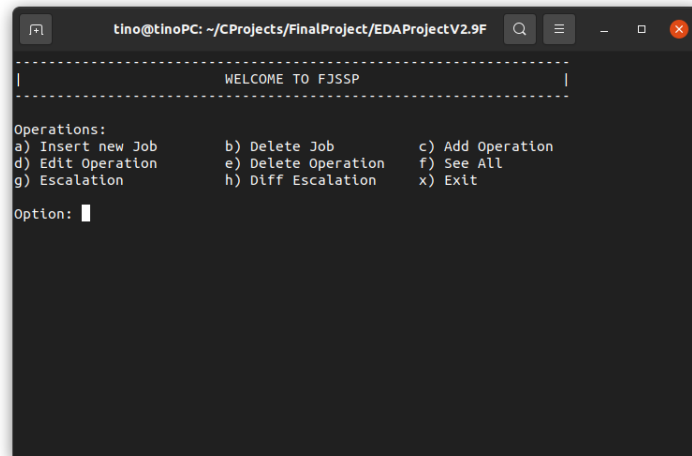
finalJb = getCombinations(jbTemp);
combJb = getBestCombinations(finalJb);
finalPPlan = getCombinationsJb(combJb);
getPossibilities(finalPPlan);

end = clock();
time_taken = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("fun() took %f seconds to execute \n", time_taken);
printf("File saved at db/bestEscalation.txt\n\n");
```

Figura 3.26: Escalonamento com localização de ficheiro

4 | Testes realizados

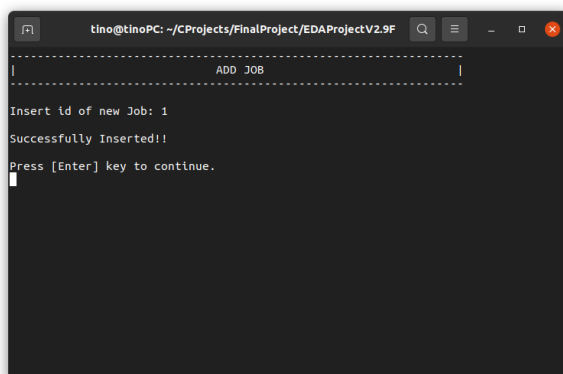
Página Principal



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
|                                     |
|                               WELCOME TO FJSSP                               |
|                                     |
| Operations:                        |
| a) Insert new Job                  b) Delete Job                  c) Add Operation          |
| d) Edit Operation                  e) Delete Operation          f) See All                |
| g) Escalation                      h) Diff Escalation             x) Exit                |
|                                     |
| Option: █                         |
|                                     |
|-----|
```

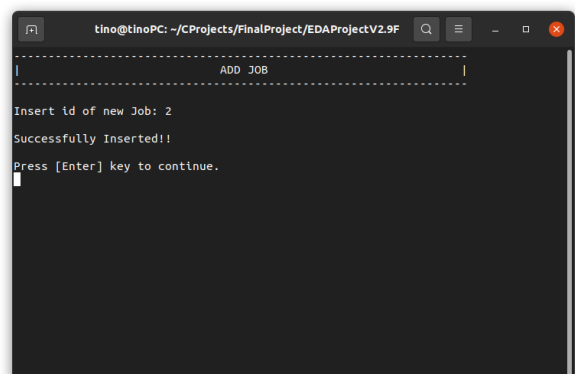
Figura 4.1: Página Principal

Exemplo de Adicionar um Job



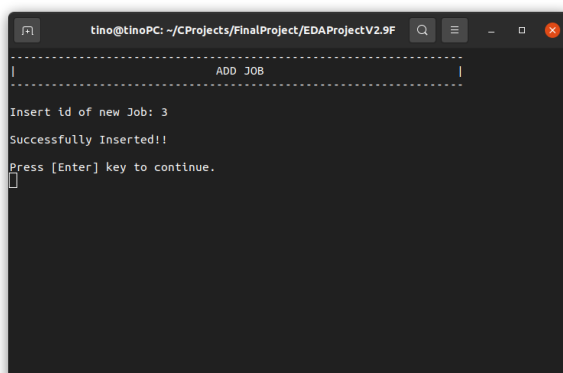
```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
|                                     |
|                               ADD JOB                               |
|                                     |
| Insert id of new Job: 1         |
| Successfully Inserted!!         |
| Press [Enter] key to continue.  |
| █                               |
|-----|
```

Figura 4.2: Adicionar



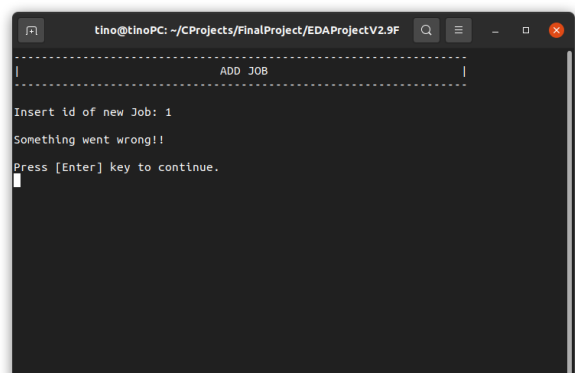
```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
|                                     |
|                               ADD JOB                               |
|                                     |
| Insert id of new Job: 2         |
| Successfully Inserted!!         |
| Press [Enter] key to continue.  |
| █                               |
|-----|
```

Figura 4.3: Adicionar



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
|                                     |
|                               ADD JOB                               |
|                                     |
| Insert id of new Job: 3         |
| Successfully Inserted!!         |
| Press [Enter] key to continue.  |
| █                               |
|-----|
```

Figura 4.4: Adicionar



```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
|                                     |
|                               ADD JOB                               |
|                                     |
| Insert id of new Job: 1         |
| Something went wrong!!         |
| Press [Enter] key to continue.  |
| █                               |
|-----|
```

Figura 4.5: Verificar

Exemplo de Remover um Job

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
|-----DELETE JOB-----|
Id
1
2
3

Insert Id to delete: 3
Successfully Deleted!!
Press [Enter] key to continue.
```

Figura 4.6: Remover

Exemplo de Adicionar uma Operação

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
|-----ADD OPERATION-----|
Id
1
2

Insert Id of Job: 1
Insert Id of operation: 1
Insert Machine number: 4
Insert time of Machine: 10
Successfully Inserted!!
Press [Enter] key to continue.
```

Figura 4.7: Adicionar

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
|-----ADD OPERATION-----|
Id
1
2

Insert Id of Job: 2
Insert Id of operation: 1
Insert Machine number: 4
Insert time of Machine: 2
Successfully Inserted!!
Press [Enter] key to continue.
```

Figura 4.8: Adicionar

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
|-----ADD OPERATION-----|
Id
1
2

Insert Id of Job: 1
Insert Id of operation: 4
Insert Machine number: 3
Insert time of Machine: 1
Something went wrong!!
Press [Enter] key to continue.
```

Figura 4.9: Adicionar

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
|-----SEE ALL JOB-----|
IdJob | IdOp | IdM | Time
1 | 1 | 4 | 10
1 | 2 | 4 | 10
1 | 2 | 5 | 3
1 | 3 | 4 | 5
2 | 1 | 4 | 2

Press [Enter] key to continue.
```

Figura 4.10: Verificar

Exemplo de Editar uma Operação

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
EDIT OPERATION
-----
IdJob | IndexOp | IdOp | IdM | Time
1 | 1 | 1 | 4 | 10
1 | 2 | 2 | 4 | 10
1 | 3 | 2 | 5 | 3
1 | 4 | 3 | 4 | 5
2 | 1 | 1 | 4 | 2

Insert id of Job: 1
Insert index to edit: 2
Insert id of operation: 4
Insert Machine number: 1
Insert time of Machine: 20
Successfully Inserted!!
Press [Enter] key to continue.
```

Figura 4.11: Editar

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
SEE ALL JOB
-----
IdJob | IdOp | IdM | Time
1 | 1 | 4 | 10
1 | 2 | 5 | 3
1 | 3 | 4 | 5
1 | 4 | 1 | 20
2 | 1 | 4 | 2

Press [Enter] key to continue.
```

Figura 4.12: Verificar

Exemplo de Remover uma Operação

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
DELETE OPERATION
-----
IdJob | IndexOp | IdOp | IdM | Time
1 | 1 | 1 | 4 | 10
1 | 2 | 2 | 5 | 3
1 | 3 | 3 | 4 | 5
1 | 4 | 4 | 1 | 20
2 | 1 | 1 | 4 | 2

Insert id of Job: 1
Insert index to delete: 4
Successfully Deleted!!
Press [Enter] key to continue.
```

Figura 4.13: Remover

```
tino@tinoPC: ~/CProjects/FinalProject/EDAProjectV2.9F
-----
SEE ALL JOB
-----
IdJob | IdOp | IdM | Time
1 | 1 | 4 | 10
1 | 2 | 5 | 3
1 | 3 | 4 | 5
2 | 1 | 4 | 2

Press [Enter] key to continue.
```

Figura 4.14: Verificar

Exemplo de Ver todas as Operações

```
tino@tinoPC: ~/CProjects/FinalProject/EDAPProjectV2.9F
-----
|                               SEE ALL JOB                               |
-----

IdJob | IdOp | IdM | Time
1 | 1 | 4 | 10
1 | 2 | 4 | 10
1 | 2 | 5 | 3
1 | 3 | 4 | 5

2 | 1 | 4 | 2

Press [Enter] key to continue.
```

Figura 4.15: Ver todas

Exemplo de ver o escalonamento

```
tino@tinoPC: ~/CProjects/FinalProject/EDAPProjectV2.9F
-----
|                               ESCALATION                               |
-----

fun() took 0.013052 seconds to execute
File saved at db/bestEscalation.txt

Press [Enter] key to continue.
```

Figura 4.16: Escalonamento

Exemplo de ver o escalonamento por localização

```
tino@tinoPC: ~/CProjects/FinalProject/EDAPProjectV2.9F
-----
|                               |
|          DIFFERENT ESCALATION          |
|                               |
-----
Insert location of file: /home/tino/CProjects/FinalProject/EDAPProjectV2.9F/db/db
.txt
fun() took 0.021676 seconds to execute
File saved at db/bestEscalation.txt
Press [Enter] key to continue.
```

Figura 4.17: Escalonamento por localização

Exemplo de sair do Programa

```
tino@tinoPC: ~/CProjects/FinalProject/EDAPProjectV2.9F
-----
|                               |
|          WELCOME TO FJSSP          |
|                               |
-----
Operations:
a) Insert new Job      b) Delete Job      c) Add Operation
d) Edit Operation     e) Delete Operation  f) See All
g) Escalation         h) Diff Escalation  x) Exit

Option: x
See you next time!

tino@tinoPC: ~/CProjects/FinalProject/EDAPProjectV2.9F$
tino@tinoPC: ~/CProjects/FinalProject/EDAPProjectV2.9F$
```

Figura 4.18: Sair

5 | Shifting Bottleneck

O método do *Shifting Bottleneck*, é um método que pode ser um bocado complicado de compreender mas aqui será explicado tudo o que é preciso para entender melhor este método.

Primeiro de tudo, explicar as palavras chaves:

- *LowerBone (LB)*: O tempo máximo que máquina demora a acabar.
- *LowerBoneMax (LBM)*: O tempo máquina demora a acabar a mais em relação ao LowerBone.
- *Processing Time (PT)*: O tempo da máquina que demora naquele Job, naquela Operação.
- *Released Time (RT)*: O tempo que se tem que esperar até poder começar, ou seja, por exemplo se for uma máquina que esteja na operação 3, esta vai ter que tempo de espera até que operação 1 e 2 acabem para esta poder começar.
- *Do Date (DD)*: O tempo restante que a máquina tem, ou seja, soma se o tempo das outras operações que faltam naquele job a partir da operação que está a maquina a ser calculada e pega se no lowerBone, e subtrai-se ambos e ai temos o nosso Do Date.
- *Completion Time (CT)*: O tempo que uma máquina demora no total até acabar, ou seja, usa se o *PT* e o *RT* e o *CT* anterior para o calcular.
- *Makespan*: O tempo total.

Para um melhor entendimento irá se usar a melhor solução que o Professor João forneceu como demonstração.

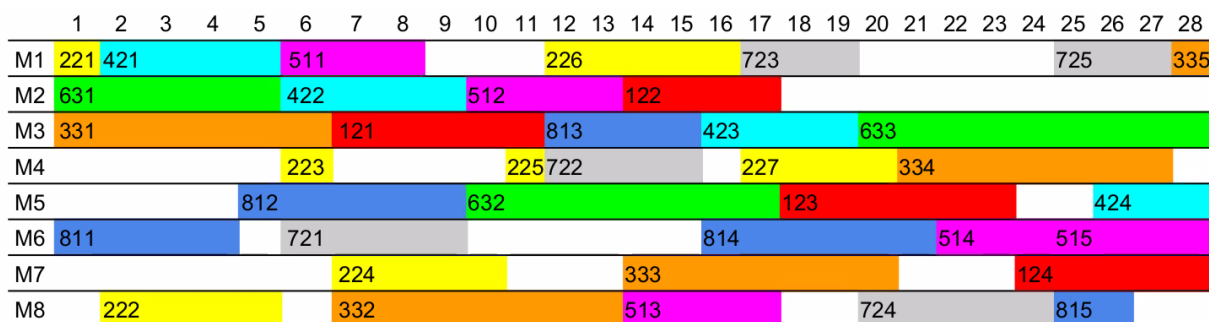


Figura 5.1: Melhor Solução

Primeiro foi se buscar o maior *LowerBone*, e que neste caso o maior *LowerBone* é a maquina 3 com 28 tempos.

	Máquina 1	Máquina 2	Máquina 3	Máquina 4	Máquina 5	Máquina 6	Máquina 7	Máquina 8
Tempos	20	17	28	17	22	21	16	22

Após descobrir o *LowerBone*, vai se buscar todos os Jobs e monta-se a tabela com *PT*, *RT* e *DD*:

Por exemplo, no job1, para calcular o *RT* como ele esta é a primeira Operação, então o seu valor é 0, por sua vez no Job 4, a máquina 3 está contida na Operação 4, portanto têm que se somar quantos tempos demora até poder ser executada, $4 + 4 + 4 = 8$ tempos.

Outro exemplo, no job1, para calcular o *DD*, como ele é a primeira Operação, têm que se somar as restantes Operações nesse job e pegar no *LowerBone* e subtrai-lo, $4 + 6 + 5 = 15$ tempos, $28 - 15 = 13$ tempos que a o job1 têm para se obter o melhor tempo.

	Job 1	Job 3	Job 4	Job 6	Job 8
PT	5	6	4	9	4
RT	0	0	8	13	9
DD	13	6	25	28	20

A partir é fazer pegar num Job e começar a organizar até se ter o *LowerBone*. Após escolher um Job, organiza-se os restantes Jobs por ordem crescente pelo seu *DD* e calcula se o seu *CT* e o *LBM*.

Para calcular *CT*, primeiro verifica-se o *RT*, se este for superior ao *CT* anterior, o novo tempo a continuar é o *RT*, pois essa Operação só pode começar quando for libertado. Como exemplo para o Job1, o *RT* é 0, portanto só se soma o seu *PT*.

No fim calcula-se o *LBM*, este é calculado quando um *CT* seja superior ao *DD*, que neste caso existe no Job3 então subtrai-se o *LBM* começando pelo Job1 é $11 - 6 = 5$ tempos.

	Job 1	Job 3	Job 8	Job 4	Job 6
CT	5	11	15	19	28
DD	13	6	20	25	28

Para o caso do Job3, este têm *LBM* de 0, o que quer dizer que é o melhor candidato para obtermos a melhor solução.

	Job 3	Job 1	Job 8	Job 4	Job 6
CT	6	11	15	19	28
DD	6	13	20	25	28

Não se calculou os restantes Jobs, pois estes têm *RT* elevados, e que o tempo que se tem que esperar até se poder começar a usar, já se podia ter feito o Job1 ou Job3.

Vai se seguir agora com o Job3, pois este tem o menor *LBM* e é o melhor candidato.

Após escolher o melhor candidato para a primeira posição, agora têm que se escolher o melhor para a segunda posição.

Verificando todos os candidatos, pode-se verificar que o melhor Job para a segunda posição é o Job1, pois os restantes têm um *RT* superior ao *PT* do Job3 portanto teriam um *LBM* maior do que usar o Job1, portanto continua-se a usar a tabela anterior.

	Job 3	Job 1	Job 8	Job 4	Job 6
CT	6	11	15	19	28
DD	6	13	20	25	28

Ao escolher o terceiro candidato, houve um empate entre o Job4 e Job8, então mantém se ambos e passa se para o quarto candidato, e também fica igual, portanto acabando assim a escolha dos melhores candidatos com estas duas opções.

	Job 3	Job 1	Job 8	Job 4	Job 6
CT	6	11	15	19	28
DD	6	13	20	25	28

	Job 3	Job 1	Job 4	Job 8	Job 6
CT	6	11	15	19	28
DD	6	13	25	20	28

No fim de encontrar a melhor solução para a maquina 3, atualizamos o valor *Makespan* com o valor *LB* + *LBM* fazemos outra vez o processo todo:

- Buscar a máquina com maior tempo;
- Ordenar os Jobs mas agora a calcular o *DD* subtraindo o *Makespan* com o tempo restante que falta para acabar o Job;
- Obter a melhor solução.

Qualquer dúvida que se possa ter, segue-se um video a explicar todos os passos com exemplos feitos durante o video, passo a passo: <https://youtu.be/RyUILLnhcKfQ>.

6 | 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.