IPCA ESCOLA SUPERIOR DE TECNOLOGIA

IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Relatório Projeto EDA – Fase 1

EDJD 2021/2022

Este projeto serviu de prepósito a um projeto para a cadeira de Estruturas de dados avançadas, em c dividido em duas fases.

Índice

Introdução

Este trabalho foi proposto pelo professor da unidade curricular Estrutura de dados Avançadas, Luis Ferreira integrada o segundo semestre do 1º ano de licenciatura de Desenvolvimento de Jogos Digitais, que visa o reforço e a aplicação dos conhecimentos adquiridos ao logo do semestre.

Propósitos e Objetivos

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. Um FJSSP pode ser formulado da seguinte forma:

1. Existe um conjunto finito de n jobs que têm de ser processados por um conjunto finito de m máquinas;
2. O conjunto de m máquinas é identificado por: M = {M1, M2, ..., Mn};
3. Um job é constituído por uma sequência de ni operações como: (Oi, 1, Oi, 2, ..., Oi, ni).
4. Cada operação deve ser executada para completar o job. A execução de cada operação j de um job i (Oi, j) requer uma máquina de um conjunto de máquinas Mi, j. O tempo de uma operação Oi, j realizada na máquina em Mi, j é pi, j, k. As seguintes suposições são consideradas num problema FJSSP:
    a. Todas as máquinas estão disponíveis no instante t = 0.
    b. Todos os jobs estão disponíveis no tempo t = 0.
    c. Cada operação pode ser realizada por apenas uma máquina de cada vez.
    d. Não há restrições de precedência entre as operações de diferentes jobs; portanto os jobs são independentes.
    e. Uma operação, uma vez iniciada, não pode ser interrompida.
    f. O tempo de transporte de jobs entre as máquinas e tempo para configurar a máquina para realizar uma determinada operação estão incluídos no tempo de processamento.

Um job é um processo de produção de uma instância de um produto específico que é definido por um *process plan*. Uma operação é uma tarefa individual que é alocada a uma máquina e está associada a um job específico. Uma máquina é um recurso capaz de executar operações, e por fim um *process plan* é uma lista ordenada de operações necessárias para concluir um job.

A Tabela 1 incorpora os *process plan* com dimensão 8x7 para a produção de um produto, envolvendo a realização de 8 jobs (com um máximo de 7 operações) distribuídos por 8 máquinas. Cada linha da Tabela 1 apresenta a descrição da sequência das operações

necessárias para cada tipo de job (um job representa a produção de um produto, por exemplo pr1,2). No caso do tipo de job pr1,2 (primeira linha do *process plan*), este requer a execução de 4 operações numa predeterminada ordem, i.e. 01, 02, 03 e 04. Para cada operação, o *process plan* indica quais são as máquinas onde a mesma pode ser realizada, bem como a respetiva quantidade de unidades de tempo necessária para a sua realização. A título de exemplo, a primeira operação (01) pode ser realizada na máquina 1 com uma duração de 4 unidades de tempo ou na máquina 3 com uma duração de 5 unidades de tempo. Cada job de um *process plan* é composto por n operações que podem ser encadeadas com outras operações de outros jobs, mas dentro do mesmo job necessitam ser executadas pela sua ordem, isto é, num job que tenha três operações, a operação 3 não pode ser iniciada sem que a operação 2 esteja finalizada, e esta por sua vez também não pode ser iniciada sem que a operação 1 esteja finalizada. O cálculo da distribuição das operações pelas máquinas terá de se basear na capacidade das máquinas poderem executar essa operação, e na ocupação destas.

Process Plan	Operation						
	01	02	03	04	05	06	07
$pr_{1,2}$	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			
$pr_{2,2}$	(1,3,5) [1,5,7]	(4,8) [5,4]	(4,6) [1,6]	(4,7,8) [4,4,7]	(4,6) [1,2]	(1,6,8) [5,6,4]	(4) [4]
$pr_{3,3}$	(2,3,8) [7,6,8]	(4,8) [7,7]	(3,5,7) [7,8,7]	(4,6) [7,8]	(1,2) [1,4]		
$pr_{4,2}$	(1,3,5) [4,3,7]	(2,8) [4,4]	(3,4,6,7) [4,5,6,7]	(5,6,8) [3,5,5]			
$pr_{5,1}$	(1) [3]	(2,4) [4,5]	(3,8) [4,4]	(5,6,8) [3,3,3]	(4,6) [5,4]		
$pr_{6,3}$	(1,2,3) [3,5,6]	(4,5) [7,8]	(3,6) [9,8]				
$pr_{7,2}$	(3,5,6) [4,5,4]	(4,7,8) [4,6,4]	(1,3,4,5) [3,3,4,5]	(4,6,8) [4,6,5]	(1,3) [3,3]		
$pr_{8,1}$	(1,2,6) [3,4,4]	(4,5,8) [6,5,4]	(3,7) [4,5]	(4,6) [4,6]	(7,8) [1,2]		

Tabela 1 - Process plan para um problema de escalonamento com dimensão 8x7 e 8 máquinas

Propósitos e Objetivos

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;

Estruturas de dados

Primeiramente para a resolução desta primeira fase, pensei em como seria feita a minha estrutura de dados para que ao longo do projeto não tenha que substituir nenhum dado nem nenhuma estrutura. Simplificando as minhas estruturas temos então que a estrutura da máquina tem somente um id e o tempo da máquina. Como é uma lista ligada tem sempre um apontador da própria estrutura de máquina para uma próxima estrutura de máquina que possa existir.

```
typedef struct Maquina{
    int id;
    int tempo;
    struct Maquina* nextMaquina;
}Maquina;

extern Maquina* headMaquina;
```

Porque me ficaria mais fácil a gestão das máquinas dentro de uma lista ligada de listas de máquinas, criei uma lista de máquinas para que numa operação tenho que ter uma ou mais máquinas. Como é uma lista ligada tem sempre um apontador da própria estrutura de listas de máquinas para uma próxima estrutura de lista de máquinas que possa existir.

```
typedef struct ListaMaquinas{
    struct Maquina maquina;
    struct ListaMaquinas* nextMaquinas;
}ListaMaquinas;

extern ListaMaquinas* headMaquinas;
```

Como anteriormente foi dito, uma operação tem uma lista de máquinas que pode ter uma ou mais máquinas e um id por operação. Como é uma lista ligada tem sempre um apontador da própria estrutura operação para uma próxima estrutura operação que possa existir.

```
typedef struct Operacao{
    int id;
    struct ListaMaquinas* maquinas;
    struct Operacao* nextOperacao;
}Operacao;

extern Operacao* headOperacao;
```

A mesma coisa acontece com as máquinas, ficaria mais fácil a gestão das operações dentro de uma lista ligada de listas de operações, criei uma lista de operações para que numa operação tenho que ter uma ou mais operações. Como é uma lista ligada tem sempre um apontador da própria de estrutura de operações para uma próxima estrutura de listas operações que possa existir.

```
typedef struct ListaOperacoes{
    struct Operacao operacao;
    struct ListaOperacoes* nextOperacoes;
}ListaOperacoes;

extern ListaOperacoes* headOperacoes;
```

Para cada *Job* tem um único id e uma lista de operações. Como é uma lista ligada tem sempre um apontador da própria de estrutura de Job para uma próxima estrutura de Job que possa existir.

```
typedef struct Job{
    int id;
    struct ListaOperacoes* operecao;
    struct Job* nextJob;
}Job;

extern Job* headJob;
```

Criei também uma estrutura para me servir de auxílio para inserir num ficheiro binário uma estrutura de *Job* sem que exista apontadores de memória tendo apenas então o id do *Job*.

```
typedef struct JobFile{
    int id;
}JobFile;
```

Testes Realizados

Para fase de testes utilizei apenas um job, uma operação e duas máquinas dentro dessa operação. A primeira tabela representa os dados inicializados por código chamando as funções para criar e inserir nas respetivas listas e na segunda tabela o tempo de uma das máquinas alterado. Para as três próximas tabelas tem o respetivo tempo mínimo e tempo máximo possível de todas as máquinas de todas as operações daquele job, assim como a média. A última tabela representa os jobs depois de eliminar uma operação desse mesmo job.

```
----------------------------------------
Jobs:
ID: 1
Operacoes:
ID: 1
Maquinas:
ID: 1 - Tempo: 4
ID: 3 - Tempo: 5


---------------------------------
Jobs:
ID: 1
Operacoes:
ID: 1
Maquinas:
ID: 1 - Tempo: 4
ID: 3 - Tempo: 8
```

```
----------------------------------------
Tempo Minimo: 4

----------------------------------------
Tempo Maximo: 8

----------------------------------------
Tempo Medio: 6.00

----------------------------------------
Jobs:
ID: 1
Operacoes:


----------------------------------------
```

Conclusão

Com esta primeira fase foi possível aprofundar mais conhecimentos acerca de estruturas de dados avançadas e também sobre listas ligadas. Aprendi também que fundamentalmente que ter um bom código e bem documentado é melhor que ter um código que funciona.