Sistemas Operativos
2021 / 2022

Licenciatura em Engenharia Informática

Trabalho Prático #2

Introdução

O Problema de Corte Unidimensional (*Cutting Stock Problem*) consiste na realização do corte de peças pequenas a partir de uma peça de tamanho maior. Este corte deverá respeitar um determinado padrão de forma a maximizar o número de peças cortadas e minimizar o desperdício.

Neste trabalho iremos assumir que as peças a cortar numa chapa de aço (denominado "placa"), são retângulos ou quadrados cuja largura é a mesma da placa. Ou seja, será apenas necessário um corte (vertical) para obter a peça desejada (fig.1).

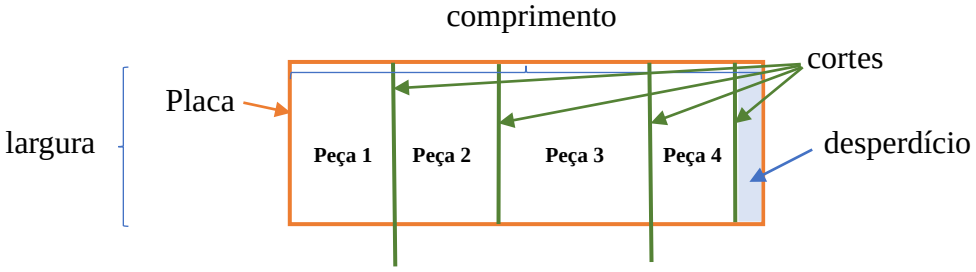

Figura 1. Exemplo de placa com peças a cortar e respetivo desperdício

Exemplo:

Sendo:

- *m* o número de diferentes comprimentos de peças em que se pretende cortar cada uma das placas
- *maxComprimento* o comprimento da placa, em metros
- *compPecas* um vetor cujos elementos representam o comprimento (em metros) das peças a cortar
- *qtddPecas*, um vetor cujos elementos representam as quantidades de peças a cortar de cada comprimento definido no vetor *compPecas*, respeitando a mesma ordem

com os seguintes dados iniciais:

- *m* = 4
- *maxComprimento* = 12
- *compPecas* = [3, 4, 5, 6]
- *qtddPecas* = [2, 2, 1, 3]

Com esta informação conseguimos saber que se pretende gerar padrões de organização das peças na placa, que a placa tem 12m de comprimento, que as peças a cortar possuem quatro comprimentos diferentes (3, 4, 5 e 6 metros), e que pretendemos obter 2 peças de 3m, 2 peças de 4m, 1 peça de 5m, e 3 peças de 6m (fig. 2).

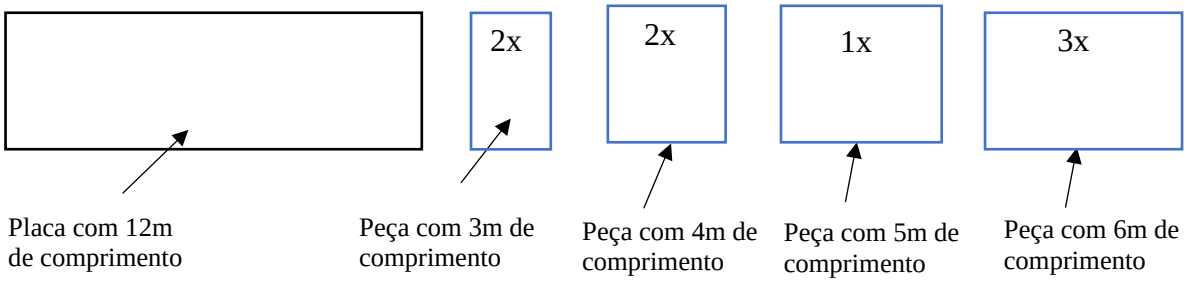

Figura 2. Placa e peças a cortar referentes ao exemplo

Na figura 3, está uma representação do problema usando um vetor ordenado, onde os comprimentos das peças (no seu número pretendido) estão representados explicitamente.

Lista items	:	5	4	6	3	3	4	6	6
Corte em	:		|		|		|	|
Desperdício	:	3		0		2		6	

Figura 3. Representação de uma solução possível

Como se pode ver pela figura, existem 4 posições de corte correspondendo a quatro placas cortadas. Na última linha está calculado o desperdício de cada placa. Sendo *w* o vetor com os valores do desperdício em cada placa, temos neste exemplo *w* = [3, 0, 2, 6].

É de realçar que, contrariamente à representação usado no algoritmo do trabalho prático anterior, nesta representação, desde que os comprimentos estejam todos inseridos (na quantidade pretendida) no vetor ordenado, todas as soluções são válidas!

Algoritmos de resolução

No trabalho prático anterior foi apresentado o algoritmo **AJR Pseudo-Evolutivo** que conseguiu apresentar alguns resultados, embora esses resultados tenham sido modestos para problemas com um maior número de peças. Neste trabalho propomos o algoritmo **AJR Evolutivo (AJR-E++)** que pretende melhorar significativamente o desempenho do algoritmo anterior.

O algoritmo AJR-E++ pertence à classe dos algoritmos evolutivos, sendo uma versão de um dos seus tipos - a Programação Evolutiva (https://en.wikipedia.org/wiki/Evolutionary_programming). Ao contrário dos algoritmos que trabalham apenas com uma solução, na programação evolutiva trabalha-se sempre com conjuntos de soluções (denominadas populações).

O algoritmo funciona da seguinte forma:

1. Inicializa-se aleatoriamente uma população de μ indivíduos. Cada indivíduo é representado por um vetor de inteiros, e deve conter todas as peças requeridas, sendo cada peça representada pelo seu comprimento.

2. Avalia-se todos os μ indivíduos da população, atribuindo a cada indivíduo um custo de acordo com a seguinte fórmula:

$$Custo = \frac{1}{n+1}\left(\sum_{j=1}^{n}\sqrt{\frac{w_j}{maxComprimento}} + \sum_{j=1}^{n}\frac{V_j}{n}\right)$$

com

$$w_j = maxComprimento - \sum_{i=1}^{m} x_{ij}\, comPecas_i\,, \quad \text{com j=1, …, n}$$

$$V_j = \begin{cases} 1 & se\ w_j > 0 \\ 0 & em\ caso\ contrário \end{cases}, \quad \text{com j=1, …, n}$$

onde

n - número de placas cortadas
w_j - desperdício da j^{ésima} placa depois de cortada
V_j - j^{ésima} placa com desperdício
x_{ij} - número de peças com comprimento $comPecas_i$ cortadas na j^{ésima} placa

3. Para cada um dos μ indivíduos (progenitores), gera-se um descendente usando a mutação *3PS* (ver anexo). Esta mutação é executada duas vezes seguidas.

4. Avalia-se o custo de cada um dos μ descendentes, usando a fórmula do ponto 2.

5. Executa-se comparações entre pares de elementos a partir do conjunto composto pela união dos progenitores e seus descendentes. Cada indivíduo deste conjunto é comparado com *q=10* outros indivíduos escolhidos de forma aleatória. Em cada comparação se o seu custo não for superior ao seu oponente ele recebe um "ganho".

6. Seleciona-se μ indivíduos entre os (μ + μ) elementos do conjunto resultante da união dos progenitores e descendentes. São selecionados os μ indivíduos com mais "ganhos" para serem os progenitores na próxima geração. Este tipo de seleção designa-se por seleção (μ + μ) dos sobreviventes.

7. O algoritmo volta ao ponto 3 enquanto não houver uma condição de término dada pelo tempo ou pelo número de iterações máxima.

No fim, o algoritmo deverá ser capaz de retornar a solução com menor avaliação (soma do número de cortes com o desperdício total) encontrada durante a sua execução. Note que a melhor solução encontrada pelo algoritmo pode ou não ser a melhor solução em termos globais.

Recomenda-se o uso de μ = 75 para o tamanho da população.

Implementação concorrencial do algoritmo AJR-E++

Dado que o algoritmo AJR-E++ tem uma forte componente aleatória, um dos grandes fatores que pode influenciar a solução é o número de iterações realizadas pelo algoritmo (ou de forma indireta, o tempo que se dá ao algoritmo para tentar encontrar a melhor solução). Desta forma, propomos a implementação paralela e concorrencial do algoritmo nas suas versões *Base* e *Avançada*.

Versão Base

1. Criar *p threads* (número parametrizável) em que cada *thread* corre o algoritmo AJR-E++.

2. Após um tempo de execução, as *threads* são interrompidas e cada uma delas atualiza a memória central com a sua melhor solução. Dado que duas ou mais *threads* podem aceder simultaneamente à memória central e corrompê-la, a atualização desta deve ser feita de forma controlada.

3. Informa-se o utilizador da melhor solução encontrada. Esta informação deve ser feita logo que **todas** as *threads* atualizem a sua melhor estratégia de corte na memória central. Para

garantir que esta atualização seja feita de forma adequada, deve ser feita a sincronização na atualização e leitura dos resultados.

Versão Avançada

A versão avançada é semelhante à versão base com as seguintes alterações:

1. De acordo com um parâmetro de entrada que representa uma percentagem do tempo total, em cada múltiplo dessa percentagem de tempo procede-se à seguinte operação:

   a) Junta-se as populações de todas as *threads* numa única população.

   b) Seja a variável μ o tamanho de cada população em cada *thread*. Ordena-se a população obtida no ponto 1.a) e escolhe-se uma nova população com as μ melhores soluções, de acordo com a avaliação.

   c) Atualiza-se todas as *threads* com a nova população obtida no ponto 1.b)

2. A operação anterior não deve ser efetuada no final do tempo de execução do algoritmo (último múltiplo da percentagem de tempo total).

Desenvolvimento

A aplicação deverá ser feita na linguagem de programação Java, em Windows (ou no seu sistema operativo preferido), usando as técnicas de programação paralela e concorrencial utilizadas nas aulas laboratoriais, nomeadamente *threads*, semáforos, métodos sincronizados, etc.

Entradas

A entrada de informação é feita usando ficheiros de texto, um para cada problema. Cada ficheiro de texto está separado por linhas, em que na primeira linha é indicado o número de comprimentos diferentes (m), na segunda linha o comprimento em metros da placa (*maxComprimento*), na terceira linha os comprimentos em metros das peças a cortar separados por um espaço (vetor *compPecas*) e na quarta linha as quantidades de peças a cortar de cada comprimento separadas por um espaço (vetor *qtddPecas*).

```
4
12
3 4 5 6
2 2 1 3
```

O programa deverá ser capaz de ser invocado passando como argumentos o nome do ficheiro de texto com o problema, o número de processos a serem criados e o tempo máximo de execução do algoritmo (em segundos). Por exemplo, o comando **pcu prob04 6 5** deverá executar o ficheiro de teste "prob04.txt" usando 6 *threads* em paralelo durante 5 segundos.

Resultados

De modo a se validar a qualidade do algoritmo, deverá ser construída uma tabela com as seguintes colunas:

a) Número do teste (de 1 a 10).
b) Nome do teste.
c) Tempo total de execução.
d) Número de processos usado (parâmetro *p* na descrição dos algoritmos).
e) Avaliação da solução (número de cortes mais valor total do desperdício).
f) Número de cortes e valor total do desperdício em metros para cada solução.
g) Número de iterações necessárias para chegar ao melhor vetor solução encontrado.
h) Tempo que demorou até o programa atingir o melhor vetor solução encontrado.

Cada teste deverá ser repetido 10 vezes para os mesmos parâmetros de entrada. Os ficheiros de teste a utilizar serão disponibilizados no Moodle da disciplina, assim como um ficheiro com alguns resultados.

Entrega e avaliação

Os trabalhos deverão ser realizados em grupos de 2 alunos da mesma turma de laboratório, e deverão ser originais. Trabalhos plagiados ou cujo código tenha sido partilhado com outros serão atribuídos nota **zero**.

Todos os ficheiros deverão ser colocados num **ficheiro zip** (com o número de todos os elementos do grupo) e submetido via *moodle* **até às 23:55 do dia 23/Janeiro/2022**. Deverá também ser colocado no zip um **relatório em pdf** com a identificação dos alunos, as tabelas de resultados e a descrições das soluções que considerarem relevantes. Este documento deverá ser mantido curto e direto (2-3 páginas).

Irá considerar-se a seguinte grelha de avaliação:

Algoritmo AJ-E++	4.0 val.
Algoritmo concorrencial	
Versão Base	4.0 val.

Versão Avançada	3.0 val.
Outra versão original	2.0 val.
Utilização de memória central	0.5 val.
Utilização de semáforos e de mecanismos de sincronização	1.5 val.
Relatório com a tabela de testes	2.0 val.
Qualidade da solução e código	3.0 val.

As discussões dos projetos serão realizadas na semana seguinte à entrega do projeto, no horário das aulas laboratoriais. As notas poderão ser atribuídas aos alunos de forma individual.

Bom trabalho!

Anexo

Mutação 3PS

Proposta por *Liang*, *Yao*, *Newton* e *Hoffman* (1999), nesta mutação são executadas trocas entre 3 posições dentro de um vetor de inteiros pertencente a um progenitor. Obtém-se no final um vetor pertencente a um descendente.

O algoritmo tem o seguinte funcionamento: A primeira posição *p1* é calculada aleatoriamente de entre as posições possíveis do vetor. A segunda e terceira posição, *p2* e *p3*, são ambas calculadas em dois passos:

1. Escolhe-se aleatoriamente, de acordo com a seguinte probabilidade, uma placa entre as *n* placas (obtidas implicitamente a partir do vetor de inteiros como na figura 3):

$$P(j) = \frac{\sqrt{1/w_j}}{\sum_{j=1}^{n} \sqrt{1/w_j}}, \forall\, w_j \neq 0$$

2. Dentro dessa j^{ésima} placa, escolhe-se de forma aleatória a posição pretendida.

Após obtidas as posições *p1*, *p2* e *p3* a mutação será feita de acordo com o seguinte pseudo-código:

```
Algoritmo swap3
Input:  v – vetor de inteiros
        p1, p2 e p3 – inteiros
Pseudo-código
        aux = v[p1]
        v[p1] = v[p2]
        v[p2] = v[p3]
        v[p3] = aux
```

Exemplo de escolha de placas de acordo com a sua probabilidade:

Sejam w_1=3, w_2=0, w_3=2, e w_4=6, os valores de desperdício (*wastage)* das 4 placas na figura 3. As suas probabilidades são

$$P_1 = \frac{\sqrt{1/3}}{\sum_{j=1}^{n} \sqrt{1/w_j}} = \frac{\sqrt{1/3}}{\sqrt{1/3}+\sqrt{1/2}+\sqrt{1/6}} = 0,3411$$

$P_2 = 0$

$$P_3 = \frac{\sqrt{1/2}}{\sum_{j=1}^{n} \sqrt{1/w_j}} = 0,4177$$

$$P_4 = \frac{\sqrt{1/6}}{\sum_{j=1}^{n} \sqrt{1/w_j}} = 0,2412$$

e as suas probabilidades acumuladas são:

$PA_1 = P_1 =$ 0,3411
$PA_2 = PA_1 + P_2 =$ 0,3411
$PA_3 = PA_2 + P_3 =$ 0,7588
$PA_4 = PA_3 + P_4 =$ 1

Se for gerado um número aleatório **r** no intervalo [0,1] podem acontecer os seguintes casos:

1. Se **r** ≤ 0,3411 escolhe-se a placa 1
2. Se 0,3411< **r** ≤ 0,7588 escolhe-se a placa 3
3. Se 0,7588 < **r** ≤ 1 escolhe-se a placa 4

Note-se que a placa 2 tem probabilidade 0, por isso nunca será selecionada.