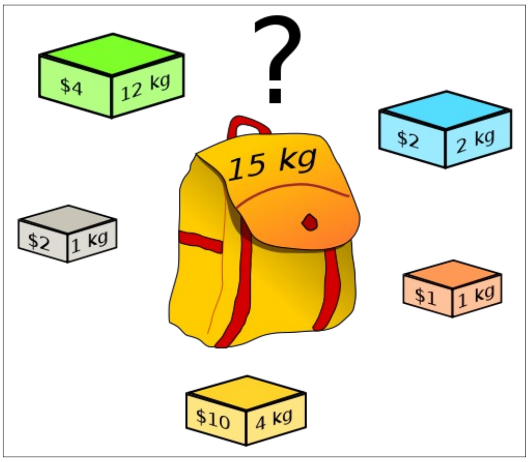
Sistemas Operativos
2019 / 2020

Licenciatura em Engenharia Informática

Trabalho Prático #2

Introdução

O problema do *Knapsack* é um problema de optimização combinatória. O seu nome tem a ver com o problema de alguém que é limitado por uma mochila de tamanho limitado e deve preenchê-la com os itens mais valiosos (https://en.wikipedia.org/wiki/Knapsack_problem).



De uma forma simples, dado um conjunto de vários itens, cada um com um determinado peso e valor, o objectivo é determinar quais os itens que devem ser incluídos numa coleção de modo a que o peso total seja menor ou igual a um determinado limite e o valor total seja o maior possível.

O problema mais comum é o *0-1 Knapsack* que restringe o número de cópias $\mathbf{x_i}$ de cada item aos valores **zero** e **um**. Dado um conjunto de $\boldsymbol{n}$ itens, cada um com um peso $\mathbf{w_i}$ e um valor $\mathbf{v_i}$, juntamente com a capacidade máxima da mochila $\mathbf{W}$, o objectivo é:

$$\textit{maximizar} \sum_{i=1}^{n} v_i x_i$$

$$\textit{sujeito a} \sum_{i=1}^{n} w_i x_i \leq W \quad \textit{com} \quad x_i \in \{0,1\}$$

Portanto o objectivo é maximizar a soma dos valores dos itens que estão dentro da mochila de modo a que a soma dos seus pesos seja menor ou igual à capacidade da mochila.

Algoritmos de resolução

No trabalho prático anterior foi apresentado o algoritmo **AJ-KPA** que apresentou, em geral, bons resultados em tempo útil para o número de items n < 25. Neste trabalho propomos o algoritmo **AJ-KP-BS** que pretende melhorar significativamente o desempenho do algoritmo anterior.

O algoritmo **AJ-KP-BS,** que usa em cada iteração o algoritmo **Beam Search** (em anexo)**,** funciona da seguinte forma:

1. Ordena-se os itens de acordo com o critério definido no anexo.

2. Calcula-se a solução de *Lower Bound* e define-se essa solução como a melhor solução até o momento.

3. Aplica-se o algoritmo *Beam Search* (BS), obtendo-se a melhor solução possível. Este algoritmo deve receber como entrada a solução de *Lower Bound* inicial.

4. Se a solução obtida no ponto 2 for melhor que a melhor solução actual, deve-se actualizar a solução actual.

5. O algoritmo volta ao ponto 2 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 o máximo valor da função de avaliação encontrado durante a sua execução. Note que a melhor solução encontrada pelo algoritmo pode ou não ser a melhor solução em termos globais.

Implementação concorrencial do algoritmo AJ-KPA

Dado que o algoritmo AJ-KP-BS terá uma forte componente aleatória, um dos grandes factores que pode influenciar a solução é o número de iterações realizadas pelo algoritmo (ou de forma indirecta, 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 *m threads* (número parametrizável) em que cada *thread* corre o algoritmo AJ-KP-BS.

2. Após um tempo de execução, as *threads* são interrompidas e cada uma delas actualiza a memória central com a sua melhor solução. Dado que duas ou mais *threads* podem aceder simultaneamente à memória central e corrompê-la, a actualização 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* actualizem a sua melhor solução na memória central. Para garantir que esta actualização seja feita de forma adequada, deve ser feita a correcta sincronização na actualização 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) Actualiza-se a melhor solução na memória central, a partir das melhores soluções de cada *thread*.

   b) Actualiza-se todas as *threads* com a nova solução obtida no ponto 1.b)

2. A operação anterior não deve ser efectuada 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 dado o número de itens ***n***, na segunda linha a capacidade máxima ***W*** da mochila, e nas restantes *n* linhas é dado um par de valores que corresponde ao valor e peso de cada item. Na última linha está o valor óptimo que se pretende obter para esse problema.

```
4
11
6 2
10 4
12 6
13 7
23
```

O programa deverá aceitar como parâmetros o nome do ficheiro de texto com o problema, o número de *threads* a serem criados e o tempo máximo de execução do algoritmo (em segundos). Por exemplo, o comando ***kp ex23.txt 10 60*** deverá executar o ficheiro de teste "ex23.txt" usando 10 *threads* em paralelo durante 60 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 e número de itens.
c) Tempo total de execução.
d) Número de *threads* usada (parâmetro *m* na descrição dos algoritmos).
e) Melhor valor da soma dos itens encontrado.
f) Valor da soma dos pesos da melhor solução.
g) Número de iterações necessárias para chegar ao melhor valor encontrado.
h) Tempo que demorou até o programa atingir o melhor valor encontrado.

Cada teste deverá ser repetido 10 vezes para os mesmos parâmetros de entrada, e deverá ser possível obter valores médios de tempo e número de iterações, assim como o número de vezes em que se encontrou o valor óptimo.

Os ficheiros de teste a utilizar serão disponibilizados no *moodle* da disciplina, assim como um exemplo de um ficheiro com resultados e respectivas estatísticas.

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 19/Janeiro/2020**. 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 directo (2-3 páginas).

Irá considerar-se a seguinte grelha de avaliação:

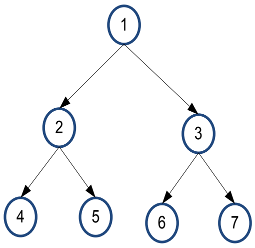
Algoritmo AJ-KP-BS	4.0 val.
Algoritmo concorrencial	
Versão Base	4.0 val.
Versão Avançada	3.0 val.
Outra versão original	2.0 val.
Utilização de memória central	0.5 val.
Utilização de mecanismos de sincronização	1.5 val.
Relatório com tabelas de testes	2.0 val.
Qualidade da solução e código	3.0 val.

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

Bom trabalho!

Algoritmo Beam Search – versão probabilística

O algoritmo *Beam Search* é um método de resolução cujo objetivo é a obtenção de soluções aproximadas num intervalo de tempo reduzido. Este algoritmo funciona com uma pesquisa em árvore onde o número dentro de cada nó indica a ordem pela qual os nós são visitados.



Dois dos conceitos mais relevantes para este algoritmo são as funções *LowerBound* (limite inferior) e *UpperBound* (limite superior). Define-se a função ***LowerBound*** (LB) como sendo a melhor solução obtida até ao momento, e a função ***UpperBound*** (UB) como aquela que estima o maior valor possível de obter a partir de um determinado nó.

A ideia básica do algoritmo é a seguinte: se o valor de *upper bound* (UB) de um determinado nó ***u*** for menor que o valor de *lower bound* (LB), então o nó ***u*** e os seus sucessores podem ser eliminados da pesquisa. Dito de outra forma, se se estimar que todas as soluções a partir do nó ***u*** nunca serão melhores que a solução actual, então não vale a pena pesquisar por soluções nessa zona.

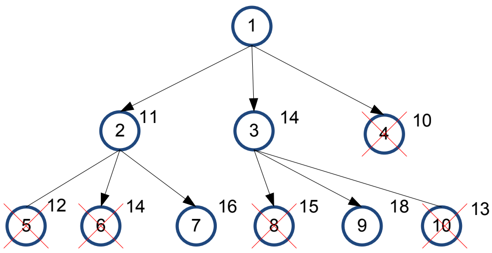
O algoritmo *Beam Search* pode ser definido pelo seguinte pseudo-código:

```
Algoritmo: Beam Search
Recebe: α, LB

 A = InitialSolution()
 Enquanto A ≠ ϕ faz
     A* = GetChilds(A)
     Para cada solução a ∈ A* faz
          ub = UpperBound(a)
          Se ub >= eval(LB) então
               Se eval(a) > eval(LB) então
                    LB = a
          senão
               A* = A* \ {a}
          Fim Se
     A = SelectSolutions(α, A*)
Devolve LB
```

A função ***GetChilds*** permite retornar uma lista de nós filhos a partir de uma lista de soluções *A*. No caso do *Knapsack*, um nó filho consiste numa solução à qual é adicionado um novo item.

A função ***SelectSolutions*** permite selecionar α nós (soluções) a partir de uma lista A* de nós. A figura seguinte mostra um exemplo onde são selecionados os α=2 elementos com melhores avaliações (o número à direita de cada nó representa a sua "avaliação"). No entanto, neste trabalho prático iremos selecionar os α nós de forma aleatória e aconselha-se o uso de α=n/2 (n é o número de items) como valor de referência.



Um ***LowerBound*** com alguma qualidade pode ser obtido inicialmente da seguinte forma:

1. Ordenar os vários items por ordem decrescente de razão v_i/w_i (valor a dividir pelo peso).
2. Ir preenchendo a solução com todos os items, de forma ordenada, até que o peso máximo seja alcançado.
3. Seja *c* a posição do primeiro item que não pode ser inserido na mochila por exceder o peso.

$$LB=\sum_{i=1}^{c-1} v_i$$

Por fim, a função ***UpperBound*** pode ser calculada da seguinte forma:

1. Ordena-se os vários items por ordem decrescente de razão v_i/w_i (valor a dividir pelo peso).
2. Seja ***s*** uma solução (incompleta) com ***k*** items já inseridos.
3. Seja **c** o índice do primeiro item que não pode ser inserido na mochila por exceder o peso.
4. Obtenha-se $\overline{W}(s)$ que corresponde ao peso que ficou de fora da solução ***s*** após a inserção de *c-1* items (i.e, peso que corresponde ao total de items fora da solução *s* com mais *c-1* items adicionados).
   a) W_{max} corresponde ao peso máximo admitido no problema.
   b) A função *sumWeigths(s)* corresponde ao somatório dos pesos dos items em ***s***.
5. O valor de *UpperBound* é obtido através da seguinte fórmula, onde a função *eval(s)* corresponde ao somatório dos valores dos items em ***s***.

$$\overline{W}(s)=W_{max}-\sum_{j=k+1}^{c-1} w_j - sumWeights(s)$$

$$UB=max\left\{\begin{array}{c} eval(s)+\sum_{j=k+1}^{c-1} v_j+\operatorname{int}\left(\overline{W}(s)\frac{v_{c+1}}{w_{c+1}}\right), \\ eval(s)+\sum_{j=k+1}^{c-1} v_j+\operatorname{int}\left(v_c-(w_c-\overline{W}(s))\frac{v_{c-1}}{w_{c-1}}\right) \end{array}\right\}$$

Exemplo

Considere o seguinte problema com 8 items.

```
8
102
60 30
40 40
90 20
15 2
100 20
15 30
1 10
10 60
280
```

Depois de ordenados por ordem decrescente de v_i/w_i, os valores e pesos ficariam na seguinte ordem:

```
v = {15, 100, 90, 60, 40, 15, 10,  1}
w = { 2,  20, 20, 30, 40, 30, 60, 10}
```

Cálculo de *Lower Bound*

Verifica-se que por esta ordem apenas se poderia inserir na mochila os primeiros 4 items (com peso 2+20+20+30 = 72), pois a inclusão do 5º item iria exceder o peso máximo (112 > 102). Portanto, o item na posição 5 é o primeiro a não poder ser inserido na mochila, e ***c*=5**.

Tendo em conta que o item na posição 5 (c=5) é o primeiro a não poder ser colocado na mochila, então o valor de ***Lower Bound*** pode ser calculado pelo seguinte somatório:

$LB = \sum_{i=1}^{c-1} v_i = \sum_{i=1}^{4} v_i$ = 15+100 +90+60 = 265

A solução *S* para o *Lower Bound* é 1111 0000 na nova ordenação (ou 1011 1000 na original).

Cálculo de um *Upper Bound*

Imagine-se agora que se está numa zona da árvore de procura com uma solução parcial **s** com 2 elementos, por exemplo, s = <10 **_ _ _ _ _ _**> na nova ordenação.

Primeiro, deve-se obter a posição do primeiro item que, ao ser colocado na mochila *s*, iria exceder o peso da mesma. Visto que em *s* o primeiro item **está** na mochila e o segundo item **não está** na mochila, a inclusão do 6º item iria exceder o peso máximo:

- s = <1 0 **1 1 1** _ _ _> tem peso 92,
- s = <1 0 **1 1 1 1** _ _> já tem peso 122, superior a 102 que é o peso máximo.

Portanto, para esta mochila *s*, o valor de ***c* = 6**.

Assim, o valor do **Upper Bound** da solução parcial **s** com 2 elementos s = <10 _ _ _ _ _ _>, sendo que **c = 6**, pode ser calculado pelas fórmulas:

$$\bar{W} = 102 - (20 + 30 + 40) - 2 = 10$$

$$UB = max\left\{eval(s) + \sum_{j=3}^{5} v_j + \text{int}\left(10 * \frac{v_7}{w_7}\right), eval(s) + \sum_{j=3}^{5} v_j + \text{int}\left(v_6 - (w_6 - 10) * \frac{v_5}{w_5}\right)\right\}$$

$$= max\left\{15 + 190 + \text{int}\left(10 * \frac{1}{60}\right), 15 + 190 + \text{int}\left(15 - (30 - 10) * \frac{40}{40}\right)\right\}$$

$$= max\{205 + 1, 205 - 5\}$$

$$= max\{206, 200\}$$

$$UB = 206$$