Lista de exercícios 2

Lucas Emanuel Resck Domingues

10 de agosto de 2021

Exercício 1 (2.5 pontos) Escolha 20 cidades brasileiras, encontre as distâncias entre elas de forma euclideana e calculando sua distância usando estradas. Mostre a execução do algoritmo A* para essas cidades (pode ser feito tanto manualmente quando implementando o algoritmo e mostrando o log de execução).

Solução: O algoritmo foi implementado para encontrar o menor caminho entre duas capitais brasileiras. O código está disponível neste Jupyter notebook. Por exemplo, calculando o melhor caminho entre Porto Alegre e Rio Branco, encontramos o seguinte log da implementação:

```
Porto Alegre    0

Porto Alegre    Florianópolis    476

Porto Alegre    Curitiba    711

Porto Alegre    Campo Grande    1518

Porto Alegre    Florianópolis    Curitiba    776

Porto Alegre    Curitiba    Campo Grande    1702

Porto Alegre    Florianópolis    Curitiba    Campo Grande    1767

Porto Alegre    Florianópolis    Campo Grande    1774

Porto Alegre    Cuiabá    2206

Porto Alegre    Campo Grande    Cuiabá    2212

Porto Alegre    Florianópolis    Porto Alegre    952

Porto Alegre    Curitiba    Cuiabá    2390

Porto Alegre    Curitiba    Campo Grande    Cuiabá    2396

Porto Alegre    São Paulo    1108

Porto Alegre    Curitiba    Florianópolis    1011

Porto Alegre    Curitiba    São Paulo    1118

Porto Alegre    Florianópolis    Curitiba    Cuiabá    2455

Porto Alegre    Florianópolis    Curitiba    Campo Grande    Cuiabá    2461

Porto Alegre    Florianópolis    Cuiabá    2462

Porto Alegre    Florianópolis    Campo Grande    Cuiabá    2468

Porto Alegre    Florianópolis    Curitiba    Florianópolis    1076
```

```
Porto Alegre    Florianópolis    São Paulo    1181

Porto Alegre    Florianópolis    Curitiba    São Paulo    1184

Porto Alegre    Curitiba    Florianópolis    Curitiba    1311

Porto Alegre    São Paulo    Campo Grande    2122

Porto Alegre    Curitiba    São Paulo    Campo Grande    2132

Porto Alegre    Florianópolis    Curitiba    Florianópolis    Curitiba    1376

Porto Alegre    Goiânia    1847

Porto Alegre    Florianópolis    São Paulo    Campo Grande    2195

Porto Alegre    Florianópolis    Curitiba    São Paulo    Campo Grande    2198

Porto Alegre    Curitiba    Goiânia    1897

Porto Alegre    Florianópolis    Curitiba    Goiânia    1962

Porto Alegre    Florianópolis    Goiânia    1969

Porto Alegre    Porto Velho    3662

Porto Alegre    Cuiabá    Porto Velho    3662

Porto Alegre    São Paulo    Curitiba    1516

Porto Alegre    Campo Grande    Porto Velho    3668

Porto Alegre    Campo Grande    Cuiabá    Porto Velho    3668

Porto Alegre    Curitiba    São Paulo    Curitiba    1526

Porto Alegre    Curitiba    Florianópolis    Curitiba    Campo Grande    2302

Porto Alegre    São Paulo    Cuiabá    2722

Porto Alegre    Curitiba    Florianópolis    Campo Grande    2309

Porto Alegre    Curitiba    São Paulo    Cuiabá    2732

Porto Alegre    São Paulo    Goiânia    2034

Porto Alegre    Curitiba    São Paulo    Goiânia    2044

Porto Alegre    Florianópolis    São Paulo    Curitiba    1589

Porto Alegre    Florianópolis    Curitiba    São Paulo    Curitiba    1592

Porto Alegre    Rio Branco    4196
```

Esses foram os nós verificados pelo algoritmo na fronteira da árvore de busca. Para cada linha, a lista mostra um histórico de caminho da origem até o nó verificado, e o número é o custo da origem até esse nó pelo caminho escolhido. Por exemplo, a última linha do log mostra que o algoritmo escolheu ir de Porto Alegre para Rio branco diretamente, com custo 4196, curiosamente. Isso significa que existe (ou deve existir) uma estrada que liga Porto Alegre a Rio Branco mas que não passa pelas outras capitais.

Um ponto interessante é que, intuitivamente, o caminho mais óbvio seria Porto Alegre, Campo Grande, Cuiabá, Porto Velho e Rio Branco. Inclusive, o algoritmo chega até Porto Velho por esse caminho e encontra um custo de 3668. Considerando que a distância por estradas entre Porto Velho e Rio Branco é de 544, então seguir por esse caminho custaria 4212. Ou seja, nosso algoritmo encontrou, realmente, um caminho de menor custo.

Exercício 2 (2.5 pontos) A tabela abaixo mostra as soluções relaxadas para todas as combinações de variáveis livres e fixas para um modelo de PLI. O modelo visa a maximização da função objetivo, com $x_1, x_2, x_3 \in \{0, 1\}$. Utilizar busca por profundidade (mas não pare na primeira solução inteira) para achar a solução ótima. Dado uma relaxação x'_1, x'_2, x'_3 , a variável x_i escolhida para ramificação será aquela cujo valor x'_i na solução relaxada esteja mais próximo de 0.5; considere também que, se os ramos possíveis forem $x_i = 0$ e $x_i = 1$, o primeiro ramo a ser explorado é aquele cujo valor esteja mais próximo de x'_i — por exemplo, dado o resultado da relaxação $(0.3, 1, 0.9)$ x_1 será explorado primeiro, começando pelo ramo $x_1 = 0$. Apresente o resultado na forma de uma árvore de busca, enumerando a sequência de exploração dos nós. Além disso, apresente uma lista descrevendo as operações realizadas em cada nó explorado (seguindo a mesma sequência de exploração).

x_1	x_2	x_3	Solução relaxação	valor
#	#	#	(0.15, 1.0, 0.05)	24.35
#	#	0	(0.05, 1.0, 0.0)	22.6
#	#	1	(0.0, 0.42, 1.0)	20.42
#	0	#	(1.0, 0.0, 0.6)	18.6
#	0	0	(0.05, 0.0, 0.0)	0.6
#	0	1	(0.6, 0.0, 1.0)	18.2
#	1	#	(0.15, 1.0, 0.05)	24.35
#	1	0	(0.05, 1.0, 0.0)	22.6
#	1	1	infactível	-
0	#	#	(0.0, 1.0, 0.2)	24.2
0	#	0	(0.0, 1.0, 0.0)	22.0
0	#	1	(0.0, 0.42, 1.0)	20.42
0	0	#	(0.0, 0.0, 1.0)	11.0
0	0	0	(0.0, 0.0, 0.0)	0.0
0	0	1	(0.0, 0.0, 1.0)	11.0
0	1	#	(0.0, 1.0, 0.2)	24.2
0	1	0	(0.0, 1.0, 0.0)	22.0
0	1	1	infactível	-
1	#	#	(1.0, 0.08, 0.47)	19.18
1	#	0	infactível	-
1	#	1	infactível	-
1	0	#	(1.0, 0.0, 0.6)	18.6
1	0	0	infactível	-
1	0	1	infactível	-
1	1	#	infactível	-
1	1	0	infactível	-
1	1	1	infactível	-

Solução: A Figura 1 apresenta a árvore resultante da busca em profundidade. Diferentemente do algoritmo original de busca em profundidade, que termina quando encontra uma solução viável, no nosso caso o algoritmo continua realizando a busca, resultando, portanto, em uma árvore.

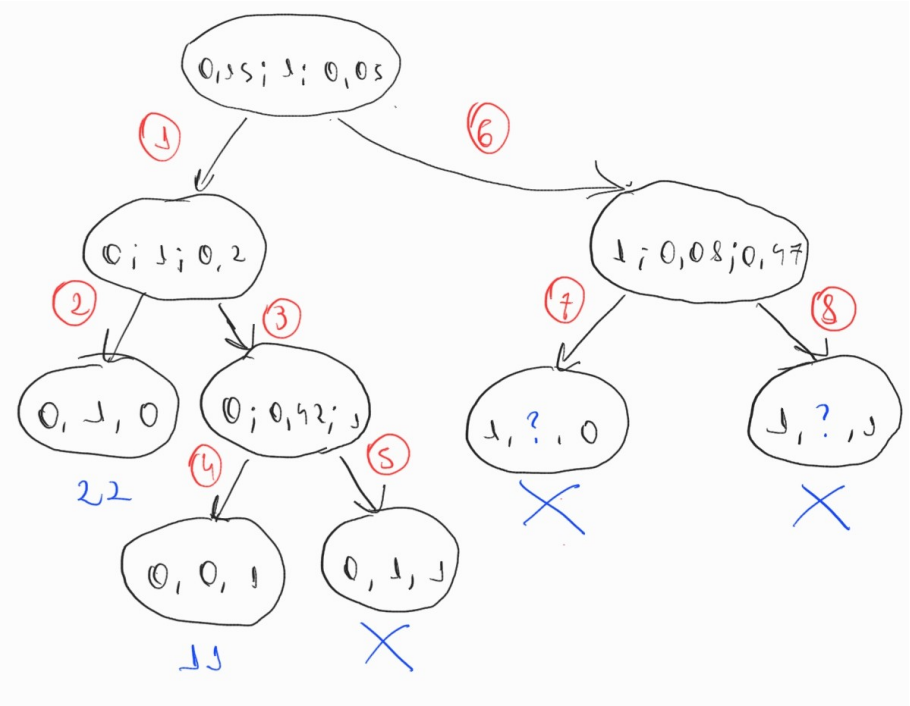

Figura 1: Cada nó da árvore apresenta uma solução relaxada. As setas indicam ramificações. Os números em vermelho indicam a ordem das operações (e fazem referência à Tabela 1. Os números em azul indicam o valor encontrado no nó. O X em azul indica uma solução infactível, de modo que não é mais possível seguir a partir dali.

Tabela 1: Lista de operações realizadas durante a busca na árvore.

Operação	**Ramifica**
1	x_1 em 0
2	x_3 em 0
3	x_3 em 1
4	x_2 em 0
5	x_2 em 1
6	x_1 em 1
7	x_3 em 0
8	x_3 em 1

Exercício 3 (2.5 pontos) Considerando o problema da mochila apresentado previamente. Vamos supor que ao invés de somente um valor de prêmio v_i para cada item da mochila, existam m valores de prêmio, e com isso teremos múltiplos objetivos. Desenhe uma busca A^* ou branch-and-bound para esse problema.

Solução: Inicialmente, vamos tratar da mochila uniobjetivo, para, posteriormente, generalizar o resultado para a mochila multiobjetivo. Para o problema da mochila original (uniobjetivo), podemos encontrar uma solução relaxada utilizando o algoritmo de Dantzig: um algoritmo guloso que enche a mochila com frações dos itens mais valiosos, ordenados pelo seu custo-benefício (valor dividido por peso), até que a mochila esteja cheia. Caso a solução resultante seja inteira, está tudo certo, e essa é a solução final.

Caso a solução para os itens não seja inteira (mais especificamente, as quantidades dos itens estejam entre 0 e w_i), precisaremos ramificar. Ramificamos no item i que possui o valor fracionário mais próximo de $0.5w_i$, e iniciamos na ramificação 0 ou w_i, dependendo de o valor relaxado estar mais próximo de 0 ou w_i. Isso descreve completamente a sequência de *branches*.

Toda vez que calculamos um novo nó após uma ramificação, também encontramos o valor da mochila uniobjetivo dessa configuração relaxada. Isso significa que, naquela subárvore, todas as soluções, inteiras ou não, têm, no máximo, esse mesmo valor. Podemos armazenar uma variável chamada *limitante inferior* de tal forma que, quando nos deparamos com uma subárvore que

tem solução ótima contínua menor do que o limitante inferior, não chegamos nem a explorar as ramificações dessa árvore. Em outras palavras, só buscamos soluções melhores do que já temos.

O algoritmo descrito acima possui forte analogia com o branch-and-bound para problemas de programação linear, pelo menos no caso uni-objetivo.

Podemos generalizar esse algoritmo para o caso multiobjetivo da mochila. A princípio, o algoritmo permanece o mesmo para o *branch* nas variáveis não inteiras. Porém, agora, vamos tomar outra estratégia para o *bound*: vamos manter as soluções multiobjetivo armazenadas em uma lista chamada *soluções não dominadas*. Quando formos decidir explorar um nó, calculamos, para cada um dos m valores dos itens, a solução ótima contínua, e reproduzimos essa configuração para os outros valores. Ou seja, para cada i, teremos uma solução multiobjetivo, porém que maximizou i apenas, e serão m soluções desse tipo. Na nossa lista *soluções não dominadas*, buscamos por uma solução que domine **todas** as nossas m soluções encontradas no nó. É importante uma mesma solução dominar todas, porque aí garantimos que essa solução domina qualquer solução intermediária, que não maximiza nenhum $0 \leq i \leq m$ particularmente. Só não vamos explorar a subárvore desse nó caso encontrarmos essa solução dominante. Se as m soluções da mochila contínua em um nó são dominadas por alguma outra solução já armazenada, essa solução armazenada tem cada uma das suas m coordenadas da solução multiobjetivo melhor do que cada uma das soluções uniobjetivo contínuas do nó. Ou seja, essa solução armazenada necessariamente domina qualquer solução multiobjetivo inteira da subárvore. Isso nos faz decidir, ou não, se exploramos a subárvore.

Toda vez que chegamos em uma folha (*leaf*), verificamos se essa solução é não dominada. Caso positivo, inserimos ela em nossa lista. Isso nos garante que, ao final do algoritmo, teremos apenas as **soluções não dominadas**, pois todas as dominadas foram descartadas.

Exercício 4 (3.0 pontos) Suponha que temos n tarefas onde cada uma toma um tempo t_i para processar em m máquinas idênticas nas quais desejamos dizer qual a sequência de tarefas em cada máquina. Uma mesma tarefa não pode ser divida entre máquinas. Para um dado planejamento, A_j é o conjunto de tarefas colocados na máquina j. Seja $L_j = \sum_{i \in A_j} t_i$ o tempo da máquina j. O tempo de execução dessas tarefas será o maior L_j dentre as máquinas. Nós consideramos um algoritmo guloso para esse problema que ordena as tarefas de tal forma que $t_1 \geq t_2 \geq \cdots \geq t_n$ e iterativamente aloca o próximo serviço na maquina com a menor carga. Demonstre quão boa é a aproximação deste algoritmo (considerando OPT é a solução ótima, esperamos algo menor que 2OPT).

Solução: Chamemos de OPT^* a solução ótima para esse problema, porém relaxando a condição de que as tarefas não podem ser divididas entre máquinas. Dessa forma, é claro que $\text{OPT}^* = \frac{1}{m}\sum_{i=1}^{n} t_i$, quando todas as máquinas possuem tempo exatamente igual. (Suponha, por absurdo, que esse não seja o caso. Então alguma máquina terá tempo menor que OPT^*, porém alguma outra máquina terá necessariamente tempo maior do que OPT^*. Ora, é possível reduzir o tempo de uma das máquinas, diminuindo o tempo de execução total. Absurdo.) É interessante pensar nesse problema como barras de altura t_i, que devem ser empilhadas em m posições diferentes, de modo a minimizar a altura da maior das m torres. Essa é a analogia que seguiremos daqui para frente.

Voltemos ao problema original. Afirmação: não existe barra que tenha sua base a partir da altura OPT^*, isto é, toda barra tem sua base iniciando em uma altura menor do que OPT^*. Ora, não é difícil ver que, caso existisse essa barra, quando ela foi colocada é necessariamente verdade que existia pelo menos uma torre com altura menor do que OPT^*, e essa barra deveria ter sido empilhada nessa torre. Para justificarmos essa segunda afirmação, podemos argumentar que, caso não houvesse torre com altura menor do que OPT^*, todas as torres teriam altura a partir de OPT^*, de modo que a média das alturas das torres fosse maior do que ou igual a OPT^*, o que é um absurdo, claramente, pois contradiz a própria definição de OPT^*.

Ora, mostramos, basicamente, que $G \leq \text{OPT}^* + \max t_i$. Daí segue que:

$$\begin{aligned} G &\leq \text{OPT}^* + \max t_i \\ &\leq \text{OPT} + \max t_i \\ &\leq \text{OPT} + \text{OPT} \\ &\leq 2\text{OPT}, \end{aligned}$$

sendo a penúltima desigualdade justificada com o fato de que, se não é possível distribuir frações das barras, então necessariamente $t_i \leq \text{OPT}$, para todo i.

Exercício 5 (1.5 pontos) Dada uma solução construída pelo algoritmo guloso do exercício anterior. Como podemos melhorar essa solução?

Solução: Um dos problemas do nosso algoritmo guloso é que não necessariamente a torre mais alta é composta pelos menores blocos, isto é, seria possível, pelo menos em alguns casos específicos, trocar blocos da torre maior com outros das torres menores, de modo a diminuir o valor da solução.

Um exemplo simples, porém que mostra essa falha específica do nosso algoritmo guloso, seria $m = 3$ e os seguintes tempos: $10, 10, 8, 8, 8, 7, 6$. Na primeira torre, teremos $10 + 7$. Na segunda, $10 + 8$. Na terceira, $8 + 8 + 6$. Ou seja, o valor dessa configuração é $8 + 8 + 6 = 22$. Porém, se trocássemos os tempos 7 da primeira torre com o tempo 8 da segunda, teremos um valor de $8 + 7 + 6 = 21$, e a primeira torre cresce de $10 + 7 = 17$ para $10 + 8 = 18$, sem prejudicar a solução.

Portanto, uma das formas de se melhorar o algoritmo é diminuir o tamanho da torre mais alta trocando barras com torres menores, no estilo de um algoritmo de **busca local**.