UFMG - PPGCC
TP2 - Paradigmas

José Leal Domingues Neto

Data: 14 de maio de 2014
PAA

1 Introdução do problema

1.1 Problema

O problema em questão é bem parecido com o caixeiro viajante: Quer-se um caminho ótimo entre cidades, que seja dividido entre dois grupos de forma a minimizar o seu custo conjunto. Mais formalmente falando, temos um grafo completo $G = (V, E)$, com cada aresta com um peso p_i. Devemos dividir os vértices em grupos $A := \{v_i, v_j, ...\}$ e $B := \{v_k, v_a, ..\}$ de forma que a soma de seus pesos seja mínima. Há a restrição de que a ordem dos vértices $v_i, v_{i+1}, ..$ deve ser respeitada ao incluir-se em um grupo.

2 Modelagens e soluções

2.1 Dinâmica

Para a resolução do problema em programação dinâmica, temos que simplesmente pensar qual é o nosso estágio e estado. O problema é que como temos dois grupos para a divisão, temos que de alguma forma incluir isso também na solução. Após pensar sobre a modelagem, como temos que passar por todas as cidades, o subproblema se limita a simplesmente a pergunta: A cidade em questão deve ser incluída no grupo *A* ou *B*? Pensado isso, a seguinte equação recursiva se forma:

$$OPT(j, A, B) = \begin{cases} 0 & \text{se } j = 0 \\ min \left\{ \begin{array}{c} P(A \cup \{c_j\}) + OPT(j-1, A - \{c_j\}, B), \\ P(B \cup \{c_j\}) + OPT(j-1, A, B - \{c_j\})) \end{array} \right\} & \text{se } j > 0 \end{cases}$$

Onde *P* retorna o peso da aresta dos dois últimos elementos do grupo. Com a função recursiva feita, temos diretamente um método estilo *bottom-up* iterativo.

2.2 Guloso

O algoritmo guloso foi relativamente mais fácil de ser construído, tendo em vista que não existem muitas possibilidades e a solução é bem mais intuitiva. O método guloso nesse caso é iterar por cada elemento de forma linear e decidir a inclusão de um vértice v_i nos grupos A ou B onde eles irão incindir o menor custo.

2.3 Backtracking

A implementação do Backtracking vai construindo todas as soluções possíveis, podando a árvore de soluções quando o seu valor atual ultrapassa o mínimo já atingido nos níveis completos. Isso é feito de forma recursiva, usando um algoritmo estilo *Depth-First-Search*, onde primeiro faz-se toda a árvore incluindo vértice v_j no Grupo *A*, checa-se a soma e a solução, e só após expande-se as opções para o grupo *B*, caso a soma atual já não ultrapasse a melhor das soluções já completas.

3 Análise da solução subótima gulosa

A solução subótima gulosa produz na maioria dos casos uma solução plausível e que poderá ser confundida como a ótima. Porém ela se engasga com alguns casos especiais. Considere o seguinte caso, onde a célula c_{ij} reflete o peso da aresta do vértice v_i para v_j para os vértices $V := \{A, B, C, D\}$

	A	**B**	**C**	**D**
A	0	1	10	11
B	1	0	12	13
C	10	12	0	14
D	11	13	14	0

Seguindo os passos do algorítmo, temos as seguintes decisões nos passos a_1 e a_2: como o vértice inicial de cada grupo não adiciona nenhum custo ao modelo, o algorítmo irá distribuir v_1 e v_2 da maneira $A = \{v_A\}$ e $B = \{v_B\}$. Porém já é possível ver, que seria interessante utilizar a aresta entre esses dois vértices, pois ela possui peso 1 apenas. No final dos passos temos a resposta: $A := \{v_A, v_C\}$ e $B := \{v_B, v_D\}$. Que não é a solução ótima.

4 Análise da complexidade de tempo e espaço

4.1 Programação Dinâmica

No algoritmo de programação dinâmica, para cada vértice v_i, será adicionado a um grupo de soluções as duas possibilidades para cada solução anterior. Significa então que dobramos nossa combinação de soluções de forma: $S := \{2^0, 2^1, ..., 2^n\}$. Temos então para essa solução a seguinte equação recursiva:

$$T(n) = T(n-1) + O(2^n)$$

Expandindo-a, temos então $\sum_{i=1}^{n} 2^i$. Sua complexidade de tempo é então $O(2^n n)$. Cada passo, somamos os elementos também, incindindo um custo $O(n)$, mas este acaba sendo irrelevante perto do anterior. Em relação a espaço, para cada passo

guardamos 2^n elementos, apagando os anteriores em cada passo. Ou seja: complexidade de espaço: $O(2^n)$.

4.2 Guloso

Como simplesmente passamos 1 vez pelo nosso grupo de vértices, decidindo localmente qual será o subgrupo assinalado, temos que o algoritmo guloso em questão é $O(n)$. Em espaço, guardamos somente nossa solução atual, sendo então $O(1)$.

4.3 Backtracking

Para o *Backtracking*, em cada passo geramos todas as possibilidades possíveis, criando uma árvore de soluções e podando-a quando necessário. Gerando a árvore temos: $O(2^n)$, já que fazemos todas as permutações. Como há a poda, a partir de certo ponto, esse número irá diminuir a cada passo. Porém não é determinístico, já que depende da entrada. O solução usa $O(2^n)$ de espaço, já que cada folha da árvore é guardado.

5 Experimentos

A seguir temos os experimentos com entradas variadas e seu respectivo runtime. Os valores estão milissegundos.

n	Backtracking	Dynamic	Greedy
4	4	2	0
8	37	15	0
14	367	216	0
18	16586	24152	1
20	34708	40120	1

6 Conclusão

Existem alguns problemas que não podem ser resolvidos em tempo polinomial. Aqui podemos ter quase certeza de que esse problema é um deles. Como vimos, os dois algoritmos, dinâmico e Backtracking são exponencias e crescem seu runtime bem rapidamente, até para entradas teoricamente pequenas. Já com 22 vértices, eu já não consegui executar o programa por completo sem dar memória extra para a JVM. Com o guloso usando heurísticas, conseguimos uma solução não muito longe da ótima com um custo muito menor. Por isso vemos que para tais problemas,

devemos nos perguntar se o caminho subótimo também não atende para a demanda em questão.