Exercícios de Algoritmos Gulosos

PARTE I

1) O algoritmo de Dijkstra tem a ideia de encontrar o nó mais próximo do vértice, depois o segundo mais próximo, depois o terceiro mais próximo, e assim por diante ... Manter um conjunto de nós explorados S para o qual nós determinamos a distância do caminho mais curto d(u) de s para u.
Algoritmo Guloso()
1. Inicialize S = {s}, d (s) = 0.
2. r(v) = min d(v)
3. Repetidamente escolha n inexplorado v que minimize
4. adicione v a S e defina d (v) = r(v).

Caso base: | S | = 1 é trivial.
Hipótese indutiva: Suponha que seja verdadeiro | S | = k >= 1. Seja v o próximo nó adicionado a S. Os mais curtos caminho de (u, v) é um caminho de comprimento r(v). Considere qualquer caminho P. Vamos ver que não é menor do que r(v). O caminho (x,y) em P que deixa S, e deixe P 'ser o sub-caminho para x. P já está muito longo assim que sai de S.

2) Seja R o conjunto de requisições e A o conjunto de requisições aceitas
Algoritmo_Guloso(Conjunto R)

```
    A ← ∅
    while(R=∅)do
        selecione uma requisição i ∈ R com o maior s(i)
        A←A∪{i}
        for toda requisição j ∈ R do
            if f(i)< s(j) then
                R←R−{j}
```

Retorne o conjunto A como o conjunto de requisições aceitas

Prova por indução:
Caso base: Sejam i1, ..., ik o conjunto de requisições em A e j1, ..., jm o conjunto de requisições em O. Deseja-se provar que k =m. Tem-se que O está ordenado e as requisições em O são compatíveis. Como o algoritmo foi implementado de modo a incluir último intervalo a iniciar, tem-se que s(j1) ≤ s(i1).
Passo indutivo: Para r > 1. Assume-se como hipótese indutiva que s(j r−1) ≤ s(i r−1). Para que o r−ésimo intervalo não comece antes, ele deve estender-se para além de s(i r). Mas isto não ocorre, pois o algoritmo tem a opção de escolher “i r” ao invés de um intervalo que comece antes. Portanto s(j r+1) ≤ s(i r+1).

3) O problema é NP-Completo e portanto um algoritmo guloso não vai conseguir fazer uma solução ótima. É possível fazer a redução para o problema da mochila, que

é conhecidamente NP-Completo. Cada intervalo tem um peso igual ao seu comprimento e valor igual ao seu valor. Há um problema em que certos itens não podem ser postos quando forem adicionados outros itens, pois são conflitantes. Agora o problema da mochila pode ser um problema de intervalos.

4) Ideia do algoritmo de Huffman: Começar com |C| folhas e realizar sequencialmente |C|−1 operações de "intercalação" de dois vértices da árvore. Cada uma destas intercalações da origem a um novo vértice interno, que sera o pai dos vértices que participaram da intercalação. A escolha do par de vértices que dar a origem a intercalação em cada passo depende da soma das frequências das folhas das subárvores com raízes nos vértices que ainda não participaram de intercala coes.

Entrada: Conjunto de caracteres C e as frequências f dos caracteres em C.
Saída: raiz de uma arvore binaria representando uma codificação ótima livre de prefixos.

1. n ← |C|

Q é a fila de prioridades dada pelas frequências dos vértices ainda não intercalados

2. Q ← C
3. para i ← 1 ate n − 1 faca
4. alocar novo registro z vértice de T
5. z.esq ← EXTRAI_MIN(Q)
6. z.dir ← EXTRAI_MIN(Q)
7. z.f ← z.esq.f + z.dir.f
8. INSERE(Q,z)
9. retorne EXTRAI_MIN(Q)

Seja T uma arvore ótima.
Sejam a e b duas folhas "irmãs" (isto é, usadas em uma intercalação) mais profundas de T e x e y as folhas de T de menor frequência.
Ideia: a partir de T, obter uma outra arvore ótima T′ com x e y sendo duas folhas "irmâs".

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

Assim, $B(T) \geq B(T')$.
Analogamente $B(T') \geq B(T'')$.
Como T é ótima, T'' é ótima e o resultado vale. □

5) Para algoritmo de Huffman para gerar código ternário, é possível utilizar a ideia de uma árvore que tem duas folhas por nível da árvore e três no último nível. A árvore

construída tera a mesma ideia da original, nos ramos mais altos ficam os símbolos, mais utilizados.

Entrada: Conjunto de caracteres C e as frequências f dos caracteres em C.
Saída: raiz de uma árvore representando uma codificação ótima livre de prefixos.

```
n ← |C|
Q é a fila de prioridades dada pelas frequências dos vértices ainda não intercalados
Q ← C
para i ← 1 ate n − 1 faca
        alocar novo registro z vértice de T
        z.esq ← EXTRAI_MIN(Q)
        z.dir ← EXTRAI_MIN(Q)
        z.f ← z.esq.f + z.dir.f
        INSERE(Q,z)
retorne EXTRAI_MIN(Q)
```

A prova é pode ser baseada na prova do exercício anterior, apenas modificando que terá três folhas no último nível.

PARTE II

1) O problema da cobertura de vértices é NP-Completo e portanto um algoritmo guloso não vai conseguir fazer uma solução ótima.

2) O algoritmo recebe como entrada um grafo G conexo com pesos e monta um grafo desconexo G’ , o qual corresponde a uma floresta de árvores composta unicamente pelos vértices de G.
Em seguida, ele ordena as arestas de G em ordem decrescente e seleciona a cada instante a de maior peso. A aresta selecionada é marcada, para não ser analisada mais tarde, e verificada se pode ser adicionada ao grafo G' de forma a não gerar ciclos.
O processo termina quando G' estiver conexo.

3) A estratégia utilizada é fazer com que usuários com um serviço é tentar minimizar o valor t, que é d(i) – s(i), em que “d” é o horário que o usuário foi servido e “s” é o horário que o usuário chegou. Portanto para resolver este problema mantêm-se uma lista para colocar o horário que o usuário i chegou. Depois para atender um usuário se procura na lista a o primeiro usuário a chegar e atende-o.