Lista de Exercícios 1 - Algoritmos I

Rita Rezende Borges de Lima
Matrícula: 2020065317

February 1, 2021

1. Não podemos afirmar que o menor caminho do grafo G, P, é o caminho mais curto do novo grafo G'. Para demonstrar que a afirmação é falsa podemos citar um contra-exemplo: Na figura 1 o caminho mais curto, P, do vértice de s até t é $\langle V_0 = \text{s, t} \rangle$, com tamanho 9. Contudo no grafo G' esse mesmo caminho possui tamanho 81, que por sua vez é maior que o caminho $\langle V_0 = \text{s, v, t} \rangle$ de tamanho 50.

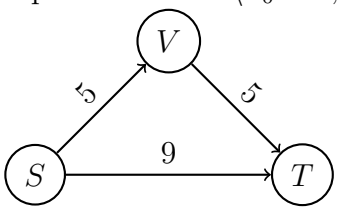

Figura 1: Grafo G

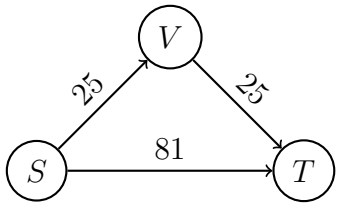

Figura 1: Grafo G'

2. (a) Uma forma de modelar o problema é utilizar um grafo não direcionado onde vértices representam pessoas e seus respectivos laços de amizade são as arestas. Para verificarmos se a teoria de que todos estão com um grau de separação de no máximo seis laços é verdade, basta verificar se o diâmetro do grafo é menor ou igual a seis.

   Uma maneira de resolver esse problema é rodar uma busca em largura para cada nó do grafo onde dentro desta primeiro zeramos o vetor auxiliar de nós visitados e depois percorremos o grafo. A busca terá uma condição de parada com o intuito de só visitar nós com distância de no máximo seis arestas. Depois da BFS, basta verificar se todos os nós foram visitados, caso contrário existem distâncias de mais de seis arestas no grafo de maneira que a teoria dos seis graus de separação é falsa.

Pseudo Código do Algoritmo apresentado previamente:

Algorithm 1: Verificar se o diâmetro do grafo é menor que seis.

```
input  : O grafo G(V,E)
output: Se todas as distâncias do grafo são menores que seis

Function BFS(ini, G(V, E)):
    Create empty queue
    foreach v ∈ G do  dist[v] ← 0 vis[v] ← false
    queue ← ini, dist[ini] ← 0
    while queue ≠ ∅ do
        u = queue.top(), queue.pop()
        foreach s ∈ G[u] do
            if vis[s] is false then
                queue.push(s)
                dist[s] = dist[u] + 1
                vis[s] ← true
                if dist[s] ≥ 7 then  return vis
    return vis

Function Main(G(V, E)):
    for v ∈ G do
        vis ← BFS(v, G)
        foreach i ∈ vis do
            if i is false then  return false
    return true
```

(b) Para o grafo G(V, E), nosso loop principal irá fazer no máximo $|V|$ iterações. Dentro do laço temos uma busca em largura de complexidade $\mathcal{O}(|V| + |E|)$ e um loop que verifica os visitados e opera no máximo $|V|$ vezes. Logo, a complexidade total é da forma $\mathcal{O}(|V|(|V| + |E|))$

3. (a) Para modelar o problema iremos considerar usuários como vértices e as influências como arestas direcionadas onde a aresta sai de quem está influenciando e o peso desta é a quantidade de retweets. Para calcularmos a influência total de um indivíduo basta calcular a soma das distâncias de todos os outros usuários em relação a este.

Exemplo: Na figura 2 João retweetou três vezes de Anna, e Caio duas, entretanto, Anna não retweetou de ninguém. No exemplo abaixo é simples perceber que Anna é a pessoa mais influente, podemos acessar qualquer outro vértice a partir dela, contudo não é possível acessá-la de nenhum outro vértice.

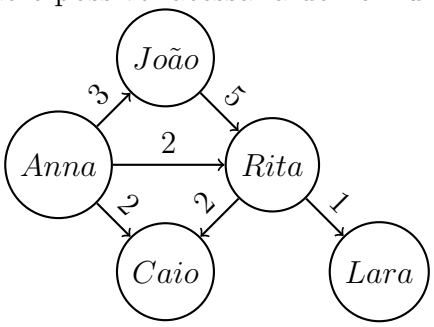

Figura 2: Exemplo da modelagem de usuários e relações

Um algoritmo que resolve este problema é calcular o vetor de distâncias por meio do algoritmo de `Dijkstra` para cada um dos vértices e depois calcular a norma um para cada vetor de distância. Usamos duas variáveis auxiliares para guardar a maior soma de distâncias e o índice do respectivo vértice e ao final retornamos o maior índice.

Algorithm 2: Retorna o vértice com maior somatório de distâncias para outros vértices

```
input  : O grafo G(V,E)
output: O vértice com maior influência

max ← 0, current ← 0, idxmax ← 0
foreach v ∈ G do
    dist ← DIJKSTRA(v, G)
    foreach i ∈ dist do  current ← current + i

    if current ≥ max then
        max ← current
        idxmax ← v
return idxmax
```

(b) O algoritmo consiste de um laço que itera cada um dos $|V|$ vértices do grafo. Dentro do loop temos o algoritmo de Dijkstra de complexidade $\mathcal{O}(|E|*\log(|V|))$ e um loop que itera para cada uma das $|V|$ posições do vetor de distância, dessa forma a complexidade geral é da forma $\mathcal{O}(|V|(|V| + |E|*\log(|V|))$ que pode ser reescrito como $\mathcal{O}(|V|(|E|*\log(|V|))$.

4. Podemos considerar influência mútua no grafo como a possibilidade de um vértice alcançar outro como visto na questão anterior. Dessa forma, grupos onde todo par tem influência mútua são na verdade componentes fortemente conexos. Um algoritmo para achar os maiores componentes fortemente conexos é o algoritmo de Kosaraju, este irá retornar os maiores grupos possíveis de usuários que possuem sempre influência mútua. O algoritmo de kosaraju é de forma simplificada duas buscas em profundidade, uma com o grafo original e outra com o grafo transposto. Computar o grafo transposto tem como complexidade $\mathcal{O}(|E|)$, a dfs tem como complexidade $\mathcal{O}(|V| + |E|)$. De forma que a complexidade total é da forma $\mathcal{O}(2(|V| + |E|) + |E|)$, ou seja, $\mathcal{O}(|V| + |E|)$.

5. (a) Uma maneira de analisar o problema é entender o terreno como um grafo G(V,E) onde todos os brinquedos e a portaria são os $|V|$ vértices, os possíveis caminhos entre estes as $|E|$ arestas e as distâncias dos caminhos os custos das respectivas arestas. O que o problema pede é encontrar o menor caminho que ligue todos os brinquedos e a portaria, o que nada mais é que encontrar a árvore geradora mínima do grafo G.

   (b) Um algoritmo que nos retorna uma árvore geradora mínima é o algoritmo de Prim. Este funciona da seguinte maneira: Iniciamos um nó qualquer como o atual corte e comparamos o valor de cada uma das arestas do atual cutset, selecionamos a menor e adicionamos esta ao conjunto de arestas da árvore geradora mínima e o outro vértice ligado a esta ao corte. Fazemos esses passos sucessivamente até termos adicionado todos os nós do grafo G ao corte. A complexidade do algoritmo de prim é a mesma de dijkstra já que a única diferença é na comparação dentro do laço, dessa forma sua complexidade é da forma $\mathcal{O}(|E|$*log$(|V|))$.

6. Para calcular o tempo gasto por um cliente da portaria até o brinquedo mais distante, podemos usar o algoritmo de Dijkstra na árvore geradora mínima retornada pelo programa da questão anterior passando o nó que representa a portaria como o nó inicial. Esta função irá retornar um vetor com as respectivas distâncias de forma que basta iterar neste em busca do maior elemento. De posse desse resultado, d, em metros e do valor médio da velocidade dos clientes, x, podemos calcular o tempo médio gasto como: y = d/x.

Algorithm 3: Retorna o maior tempo gasto em caminhada no parque

input : O grafo G(V,E), a portaria e a velocidade média dos clientes, x
output: O maior tempo necessário para chegar em um brinquedo

```
max ← 0, current ← 0
dist ← DIJKSTRA(beginning, G)
foreach i ∈ dist do
    if i ≥ max then max ← i
return max / x
```

7. Podemos considerar os brinquedos e a portaria como os vértices do grafo G(V, E), e os trilhos do trem como as arestas direcionadas. O problema consiste de verificar se o grafo G é fortemente conexo. Para isso podemos utilizar o algoritmo de Kosaraju. Este computa uma busca em profundidade sucessivamente até que tenha percorrido o grafo inteiro e calculado os tempos de descoberta e término. Depois, mais uma busca em profundidade é feita, contudo, no grafo transposto. Dessa vez escolhemos o vértice inicial considerando como critério o nó com o maior tempo de término que ainda não foi visitado pela segunda dfs. A cada vez que a dfs no grafo transposto termina um componente fortemente conexo foi completamente percorrido e iremos começar outro. Logo, para sabermos se dois vértices não possuem acessibilidade mútua, basta verificar se a função retornou mais de um componente fortemente conexo. Nessa caso, um vértice não conseguirá acessar outro, ou seja, na malha projetada não é possível acessar todo brinquedo a partir de qualquer outro de maneira que esta não está correta. O algoritmo proposto tem complexidade $\mathcal{O}(|E| + |V|)$.