Prova 1

Bruno M. Pacheco (16100865)
Grafos

11 de Março de 2021

Exercise 1

Podemos utilizar uma modificação do algoritmo de busca em largura (Algoritmo 3 das anotações), que já utiliza essa abordagem de varrer o grafo pelos vizinhos do vértice de origem. Não precisamos mais registrar a distância nem o predecessor. Entretanto, precisamos controlar os níveis de cada vértice em relação ao original, uma vez que denotam o instante de tempo em que foram contaminados. Para isso, ao invés de varrer as distâncias ao final, podemos controlar os recém infectados (Q_{i+1}). O algoritmo de busca em largura modificado pode ser observado abaixo.

Input: um grafo $G = (V, E)$, vértice $x \in V$, inteiro k
$C_v \leftarrow$ false $\forall v \in V$
$C_x \leftarrow$ true
// Mantemos controle da propagação por unidade de tempo
$Q_0 \leftarrow$ Fila()
Q_0.enqueue(x)
// Conterá as unidades de tempo em que houveram mais do que k infecções
$K \leftarrow$ conjuntoVazio()
$i \leftarrow 0$
while Q_i*.empty()*$=$ *false* **do**
    $i \leftarrow i + 1$
    $Q_i \leftarrow$ Fila()
    foreach $u \in Q_{i-1}$ **do**
        foreach $v \in N(u)$ **do**
            if $C_v =$ *false* **then**
                $C_v \leftarrow$ true
                Q_i.enqueue(v)
    if $|Q_i| \geq k$ **then**
        K.adiciona(i)
return K

Apesar de adicionar um laço, não impactamos a quantidade de iterações em cima dos vértices, uma vez que esse laço se faz presente somente para separar as unidades de tempo. Portanto, mantemos a mesma complexidade computacional do algoritmo de busca em largura, ou seja, $O(|V| + |E|)$.

Exercise 2

Com as informações fornecidas da propriedade, podemos montar um grafo completamente conectado em que cada vértice representa um ponto de aferição ou a guarita e as arestas são valoradas pelas distâncias entre as localidades de interesse. Dessa forma, o algoritmo de Bellmann-Held-Karp como definido no Algoritmo 7 das anotações, nos retorna a distância total percorrida no caminho mínimo. Dessa forma, nos basta verificar quais dos funcionários são rápidos o suficiente. A implementação pode seguir o descrito no algoritmo abaixo.

Input: conjunto de funcionários F, função v, conjunto A, ponto g, função d, valor k

// Monta o grafo dos pontos pelos quais os guardas passam

$V \leftarrow A \cup \{g\}$

$E \leftarrow V \times V$

foreach $\{u, v\} \in E$ **do**

  $w(\{u, v\}) \leftarrow \|d(u) - d(v)\|$

// c a distância do ciclo hamiltoniano mínimo

$c \leftarrow$ Bellmann-Held-Karp(G)

$F_v \leftarrow$ conjuntoVazio()

foreach $f \in F$ **do**

  if $\frac{c}{v(f)} < k$ **then**

    F_v.adiciona(f)

return F_v

Exercise 3

Podemos abordar esse problema utilizando o algoritmo de Dijkstra conforme Algoritmo 11 das anotações. Usaremos ele para encontrar os caminhos mínimos de s até os vértices de P e desses para todos os vértices de I. Para todo $i \in I$, como o grafo não é dirigido, podemos encontrar o caminho de todos os postos para i simplesmente invertendo o caminho mínimo de i para os postos, que pode ser obtido aplicando Dijkstra com i como fonte. A partir disso, basta encontrar o posto de combustível tal que a soma dos caminhos mínimos de s para p e de p para i seja mínima.

Essa abordagem pode ser vista no algoritmo abaixo. Note o uso da função "caminho", que indica a construção de um caminho usando o vetor de antecessores conforme resultante do algoritmo de Dijkstra, com um dado destino. Além disso, a função "reverte"é uma simples inversão do caminho de entrada.

Input: um grafo $G = (V, A, w)$, vértice $s \in V$, $P \subseteq V$, $I \subseteq V$

$D^{(s)}, A^{(s)} \leftarrow$ Dijkstra(G, s)

$D_i \leftarrow \infty \, \forall i \in I$

$p_i \leftarrow$ null $\forall i \in I$

foreach $i \in I$ **do**

  $D^{(i)}, A^{(i)} \leftarrow$ Dijkstra(G, i)

  $V_i \leftarrow G_i \leftarrow ()$ **foreach** $p \in P$ **do**

    if $D_p^{(s)} + D_p^{(i)} < D_i$ **then**

      $D_i \leftarrow D_p^{(s)} + D_p^{(i)}$

      $p_i \leftarrow p$

  $c_i^{(s)} \leftarrow$ caminho$\left(A^{(s)}, p\right)$

  $c_i^{(i)} \leftarrow$ caminho$\left(A^{(i)}, p\right)$

  $c_i \leftarrow c_i^{(s)} +$ reverte$\left(c_i^{(i)}\right)$

return c

Exercise 4

Podemos modificar o algoritmo de Floyd-Warshall apresentado no Algoritmo 12 das anotações para computar a quantidade de arcos do caminho mínimo durante os relaxamentos, ou seja, cada vez que há um relaxamento nós atualizamos a quantidade de arcos no caminho. Para isso, podemos utilizar uma nova matriz de adjacências. Uma implementação possível pode ser vista no algoritmo abaixo.

Input: um grafo $G = (V, E, w)$
$D^{(0)} \leftarrow W(G)$
// G_a o grafo com arestas de peso unitário, para gerar a matriz A
$w_a((u, v)) \leftarrow 1 \, \forall \, (u, v) \in E$
$G_a \leftarrow (V, E, w_a)$ $A^{(0)} \leftarrow W(G_a)$
foreach $k \in V$ **do**
    $D^{(k)} \leftarrow D^{(k-1)}$, uma cópia
    $A^{(k)} \leftarrow A^{(k-1)}$, uma cópia
    foreach $u \in V$ **do**
        foreach $v \in V$ **do**
            if $d_{uk}^{(k-1)} + d_{kv}^{(k-1)} < d_{uv}^{(k)}$ **then**
                $d_{uv}^{(k)} \leftarrow d_{uk}^{(k-1)} + d_{kv}^{(k-1)}$
                $a_{uv}^{(k)} \leftarrow a_{uk}^{(k-1)} + a_{kv}^{(k-1)}$
return $\left(D^{(|V|)}, A^{(|V|)}\right)$

Dessa forma, o algoritmo retorna não só o custo dos caminhos mínimos entre quaisquer dois vértices através da matriz D como também a quantidade de arcos no caminho mínimo através da matriz A, uma vez que essa é "relaxada"(nem sempre diminui a quantidade de arcos) somente quando a matriz D o é.