Grafos - Menor Caminho

Tópicos Especiais em Algoritmos - Ciência da Computação

INSTITUTO FEDERAL
Brasília

Prof. Daniel Saad Nogueira Nunes

IFB – Instituto Federal de Brasília, Campus Taguatinga

Sumário

Sumário

Menor Caminho

- Detectar o menor caminho por dois vértices é fácil quando o grafo não possui peso.
  - Busca em largura.
- Se o grafo possuir pesos, como resolvemos este problema?
- Para iniciar esta discussão, precisamos examinar alguns conceitos.

Menor Caminho

Definição (Custo de um Caminho)

- Suponha um grafo dirigido $G(V, E)$ e uma função de peso sobre as arestas $w : E \to \mathbb{R}$.
- Seja $P = (v_0, \ldots, v_{k-1})$ os vértices que fazem parte de um caminho. O custo do caminho é definido como:

$$w(P) = \sum_{i=0}^{k-2} w(v_i, v_{i+1})$$

Menor Caminho

Definição (Alcançabilidade)

- Suponha um grafo dirigido $G(V, E)$ e uma função de peso sobre as arestas $w : E \to \mathbb{R}$.
- Seja $P = (v_0, \ldots, v_{k-1})$ os vértices que fazem parte de um caminho com $u = v_0$ e $v = v_{k-1}$.
- Neste caso, dizemos que $u \to^P v$, isto é, existe um caminho de u para v.

Menor Caminho

Definição (Custo do Menor Caminho)

- O custo do menor caminho de um vértice u para um vértice v é dado por:

$$\delta(u,v) = \begin{cases} \min\{w(P) | u \rightarrow^{P} v\}, & \text{se existe um caminho de } u \text{ a } v \\ \infty, & \textbf{caso contrário.} \end{cases}$$

Menor Caminho

- Agora podemos definir o problema!

Menor Caminho (SSSP)

- Entrada: um grafo dirigido $G(V, E)$, uma função de peso $w : E \to \mathbb{R}$ e um vértice de origem s.
- Saída: $\delta(s, v)$, o custo do menor caminho de s até os demais vértices v.

Menor Caminho

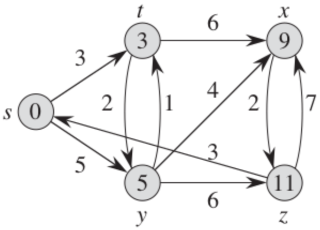


Figura: Grafo $G(V,E)$ com as respectivas distâncias de uma origem.

Menor Caminho

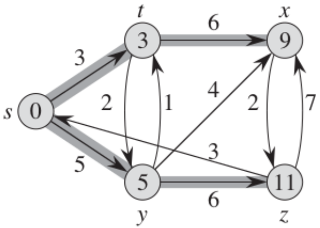


Figura: Menor rota até um destino.

Menor Caminho

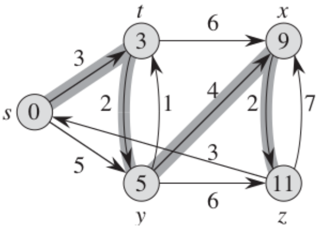


Figura: Menor rota até um destino.

Menor Caminho

- Veremos agora uma série de algoritmos que resolve o problema do menor caminho.

Sumário

Algoritmo de Dijkstra

- O algoritmo de Dijkstra se parece muito com uma busca em largura.
- Só que em vez de pegar sempre o próximo vizinho, consideramos o nó com menor custo até o momento.
- Pode ser visto como um algoritmo guloso!

Algoritmo de Dijkstra

- O algoritmo de Dijkstra se baseia no “relaxamento” de distâncias até chegar na distância ótima.
- Se a distância atual de uma origem a um nó v é maior do que a distância atual da origem a um nó u mais $w(u, v)$, atualizamos a distância da origem até v.

$$v.d > u.d + w(u, v)$$
$$v.d \leftarrow u.d + w(u, v)$$

Algoritmo de Dijkstra

Algorithm 1: $\textsc{Initialize-SSSP}(G, s)$

Input: G, s

for all($v \in V$)
- $v.d \leftarrow \infty$
- $v.\pi \leftarrow$ **NULL**
- $v.color \leftarrow$ **white**

$s.d \leftarrow 0$

Algorittmo de Dijkstra

Algorithm 2: DIJKSTRA(G, s, w)

Input: G, s, w

Output: $\delta(s, v), \forall v \in V$

```
INITIALIZE-DJIKSTRA(G, s)
Q.INSERT(s)
while ¬Q.EMPTY do
    u ← Q.EXTRACT-MIN()
    u.color ← black
    for all( (u, v) ∈ E ∧ (v.color = white) )
        if( v.d > u.d + w(u, v) )
            v.d ← u.d + w(u, v)
            v.π ← u
            Q.INSERT-UPDATE(v) // Insere ou atualiza as
                informações do nó na estrutura Q
```

Algoritmo de Djikstra

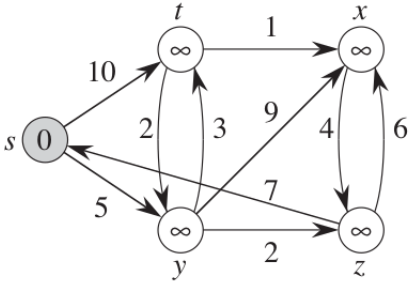


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

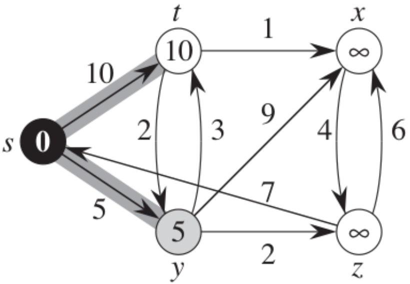


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

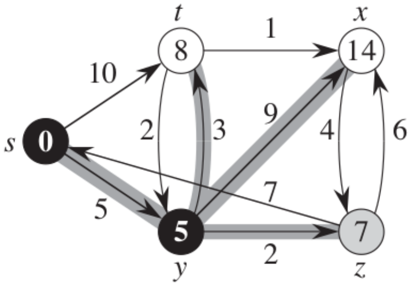


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

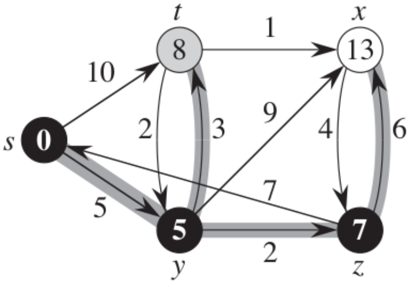


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

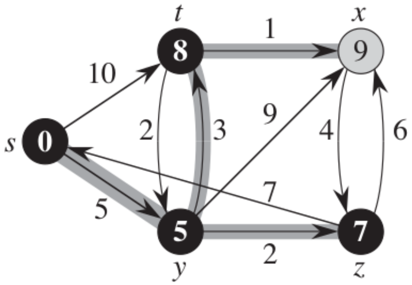


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

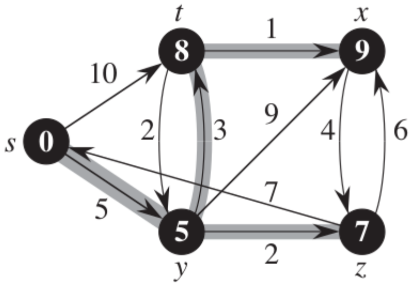


Figura: Algoritmo de Dijkstra.

Algoritmo de Djikstra

Complexidade

- Qual a complexidade do algoritmo?
- Depende da estrutura Q utilizada.
- **For** interno: $\Theta(|E|)$ vezes.
- **While** externo: $\Theta(|V|)$ vezes.
- Vai depender do custo das operações EXTRACT-MIN e INSERT-UPDATE para a estrutura de dados Q.

Algoritmo de Djikstra

Complexidade

- Usando vetores: $\Theta(|V|^2 + |E|)$.
- Usando heap: $\Theta(|V| \log |V| + |E| \log |V|)$.
- Usando heap de fibonacci: $\Theta(|V| \log |V| + |E|)$.
- O que você vai usar em grafos densos? E em grafos esparsos?

Algoritmo de Dijkstra

Limitações

- Apesar de ser um algoritmo clássico, o algoritmo de Dijkstra apresenta alguns problemas.
- Se a função w atribuir um custo negativo às arestas, o algoritmo de Dijkstra não apresentará o comportamento esperado.

Sumário

Bellman-Ford

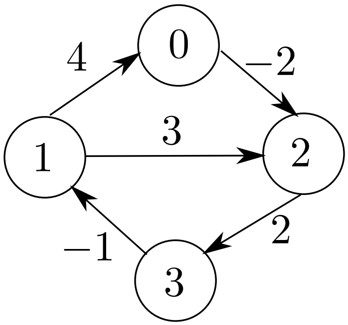
- O algoritmo de Bellman-Ford é um algoritmo que encontra o menor caminho de um vértice s para os demais vértices de um grafo.
- Ao contrário do algoritmo de Dijkstra, o algoritmo de Bellman-Ford oferece suporte para grafos com peso negativo nas arestas.
- Problema: se existem ciclos de custo negativo, o algoritmo de Bellman-Ford não dá a resposta correta, mas ele indica que existem ciclos de custo negativo.

Bellman-Ford

- A ideia básica do algoritmo de Bellman-Ford é relaxar todas as $|E|$ arestas do grafo.
- Caso esse processo seja repetido $|V|-1$ vezes, todos os caminhos mais curtos a partir de um vértice de origem são computados corretamente, pois qualquer caminho possui no máximo $|V|-1$ vértices.
- A intuição nos diz que a na iteração i pelo menos o menor caminho de um vértice fonte aos outros vértices utilizando no máximo i arestas é obtido.

Bellman-Ford

Exemplo



Bellman-Ford

Algorithm 3: BELLMAN-FORD(G, s, w)

Input: G, s, w

Output: **true** se e somente se $\delta(s, v)$ foi computado corretamente $\forall v \in V$

INITIALIZE-SSSP(G, s)
for($i \leftarrow 0; i < |V| - 1; i++$)
    for all($(u, v) \in E$)
        if($u.d + w(u, v) < v.d$)
            $v.d \leftarrow u.d + w(u, v)$ `// relaxamento`
            $v.\pi \leftarrow u$
for all($(u, v) \in E$)
    if($v.d > u.d + w(u, v)$)
        return false `// Existência de ciclos negativos`
return true

Bellman-Ford

Complexidade

- $\Theta(|V| \cdot |E|)$.
- Mais custoso que o Dijkstra, mas consegue lidar com grafos com peso negativo de arestas.
- Realiza detecção de ciclos negativos.

Bellman-Ford

Checagem de Ciclos Negativos

- A checagem de ciclos negativos funciona pois, após computar $v.d$ para todo $v \in V$, nenhuma melhoria deveria ser possível após uma nova iteração de relaxamento.

Sumário

Floyd-Warshall

- Diferentemente dos algoritmos anteriores, o algoritmo de Floyd-Warshall se preocupa em calcular a menor distância entre todos os pares de vértices (`APSP`).
- Ele se baseia em um argumento de programação dinâmica para computar a menor distância entre qualquer par de vértices em tempo eficiente.
- Funciona para grafos com arestas negativas, mas não para grafos com ciclos negativos, apesar de ser possível detectá-los.

Floyd-Warshall

APSP

- Entrada: $G = (V, E)$ um grafo dirigido e $w : E \rightarrow \mathbb{R}$ uma função de peso nas arestas.
- Saída: $\delta(u, v)$, $\forall u \forall v \in V$.

Floyd-Warshall

- Seja $V = \{v_0, \ldots, v_{n-1}\}$ o conjunto de vértices.
- $T(u, v, k)$ nos fornece a menor distância $\delta(u, v)$ utilizando apenas como vértices intermediários aqueles presentes em $\{v_0, \ldots, v_{k-1}\}$.
- Existem duas opções: o vértice v_{k-1} está no menor caminho entre u e v, considerando apenas primeiros k vértices de V como intermediários, ou não está.
- No segundo caso, temos $T(u, v, k) = T(u, v, k-1)$.
- No primeiro:

$$T(u, v, k) = T(u, k, k-1) + T(k, v, k-1)$$

Floyd-Warshall

Relação de Recorrência

$$T(u,v,k) = \begin{cases} \infty, \quad k = 0 \wedge (u,v) \notin E \\ 0, \quad k = 0 \wedge u = v \\ w(u,v), \quad k = 0 \wedge (u,v) \in E \\ \min\{T(u,v,k-1), \\ \quad T(u,k,k-1) + T(k,v,k-1)\}, \quad k > 0 \end{cases}$$

Floyd-Warshall

- Estamos interessados na resposta em computar $T(u, v, k)$, para $k = |V|$ e todo par de vértices u e v.
- Como só estamos interessados na situação em que $k = |V|$, podemos usar uma abordagem *bottom-up* de programação dinâmica e utilizar uma estrutura bidimensional, e não tridimensional, para armazenar a solução dos subproblemas.

Floyd-Warshall

Algorithm 4: INITIALIZE-FLOYD-WARSHALL(G, w)

Input: G, w

Output: Inicialização de DP para o algoritmo de Floyd-Warshall

```
for all( (u, v) ∈ V × V )
    DP[u][v] ← ∞
for all( (u, v) ∈ E )
    DP[u][v] ← w(u, v)
for all( u ∈ V )
    DP[u][u] ← 0
return DP
```

Floyd-Warshall

Algorithm 5: $\textsc{Floyd-Warshall}(G, w)$

Input: G, w

Output: $\delta(u, v), \forall u \forall v \in V$

$DP \leftarrow \textsc{Initialize-Floyd-Warshall}(G, w)$

for($k \leftarrow 1; k \leq |V|; k++$)

    for($u \leftarrow 0; u < |V|; u++$)

        for($v \leftarrow 0; v < |V|; v++$)

            if($DP[u][k] + DP[k][u] < DP[u][v]$)

                $DP[u][v] \leftarrow DP[k][u] + DP[k][v]$

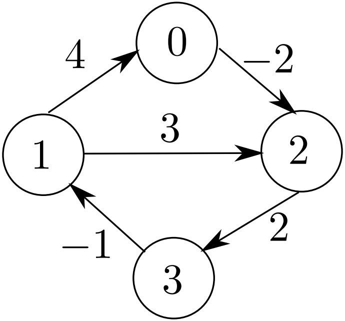
Floyd-Warshall

Complexidade

- $\Theta(|V|^3)$.

Floyd-Warshall

Exemplo



Floyd-Warshall

Reconstrução do Caminho

- Através do algoritmo de Floyd-Warshall também é possível reconstruir caminho de menor custo.
- Precisamos de uma outra matriz $\Pi[u][v]$ que armazenará o próximo nó pertencente ao menor caminho $u \to^P v$.
- Na inicialização $\Pi[u][v] \leftarrow v$ para cada aresta $(u, v) \in E$.
- Durante o algoritmo de programação dinâmica, $\Pi[u][v] \leftarrow k$ sempre que $DP[u][v] > DP[u][k] + DP[k][v]$.

Floyd-Warshall

- Para o algoritmo de programação dinâmica, é conveniente utilizar a matriz de adjacências como o espaço destinado à tabela de programação dinâmica, pois ela já possui boa parte das informações necessárias durante o procedimento $\textsc{initialize-floyd-warshall}(G, w)$.

Sumário

Considerações

- Vimos diferentes algoritmos para tratar do problema do menor caminho partindo de um nó fonte (`SSSP`) ou entre todos os pares de vértice (`APSP`).
- Estes algoritmos possuem um *trade-off* no que tange o tempo e a capacidade de lidar arestas de peso negativo.