FTP_Alg
Graphs, Breadth–First Search, Single–Source Shortest Paths, Dijkstra's Algorithm

jungkyu.canci@hslu.ch

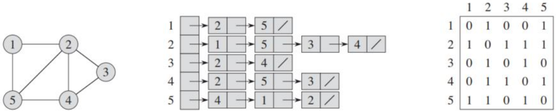
21. October 2024

Introduction

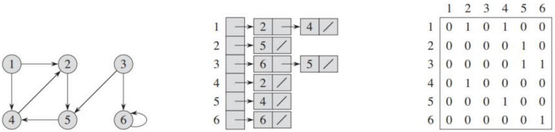
- In these slides we present Dijkstra's algorithm, that is an example of a so called Greedy algorithm.
- A **Greedy algorithm** is characterized by the property that at each stage of the algorithm, the optimal choice is made.
- Dijkstra's algorithm solves the **Single–Source Shortest Paths**. In a graph (ex. points in a city linked with routes), from a vertex s and for any other vertex v we want to determine the shortest-path-weight joining s to v.
- We start by presenting how to represent graphs.
- In the last part we will present Dijkstra's algorithm.

Representations of graphs

- A Graph is the data of two sets: V the vertices and E the edges joining some vertices in V. There are some graphs where the edges are directed and some graphs where the edges are undirected.
- In both cases, there are two standard ways to present graphs: as a collection of adjacency list or as an adjacency matrix.



(a) (b) (c)



(a) (b) (c)

Adjacency–list representation

- In the **adjacency–list representation** we have for each vertex of $u \in V$ a list with all vertexes v such that there is an edge $(u, v) \in E$. This list is denoted by $Adj[u]$. There are exactly $|V|$ adjacency lists.
- Thus in a graph $G = (V, E)$, $Adj[u]$ is the list of all vertexes of G adjacent to u. In pseudocode this will be denoted by $G.Adj[u]$.
- In a undirected graph, the sum of all lenght of the adjacency lists is $2|E|$. In a directed graph this sum is $|E|$.
- An adjacency list needs a storage of $\Theta(|V| + |E|)$.
- A **weighted graph** is a graph $G = (V, E)$ equipped with a **weight** (or **weight function**) $\omega\colon E \to \mathbb{R}$ (which measures a *distance* between two adjacent vertices).

Adjacency-matrix representation

- We assume that in the graph $G = (V, E)$, the vertices are numbered $1, 2, \dots, |V|$.
- The adjacency matrix is a $|V| \times |V|$ matrix $A = (a_{ij})$ with

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- An adjacency matrix needs $\Theta(|V|^2)$ storage (independent on $|E|$).
- In the case of an undirected graph, the matrix A is symmetric (i.e $A = A^T$, that is A is equal to its transpose).
- An adjacency matrix can also represent a weighted graph. The entry a_{ij} is the weight of the edge (i,j). In the case the edge $(i,j) \notin E$, then we can attribute to a_{ij} the value NIL. But often it is better to use the value ∞ (or 0).

Representing attributes

- We will need that our algorithms maintain some attributes, which will be denoted in the usual way.
- For example an attribute will be the *distance* d and we will denote by $v.d$ the attribute distance of the vertex v from a source $s \in V$.
- Edges can have attributes as well. If f is an attribute, we denote by $(u, v).f$ the attribute f of the edge (u, v).
- If we denote the vertices of a graph with an array with indexes $1, \ldots, |V|$, then $d[1, \ldots, |V|]$ will denote the parallel array of the attributes d. Similarly for an array $Adj[u]$ and arrays of edges.

Breadth-first search (optional part)

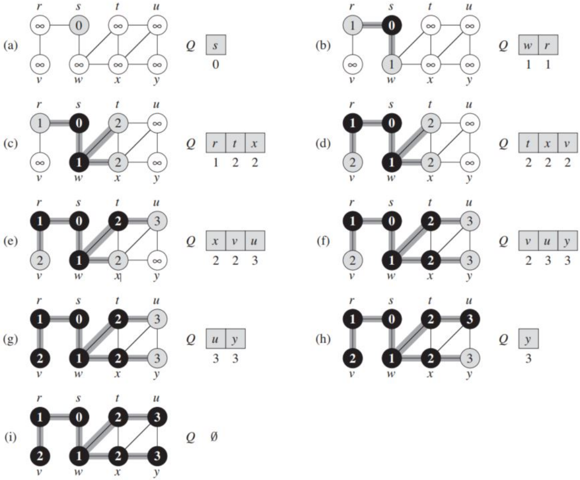
- **Breadth-first search** (BFS) is one of the simplest algorithm for searching a graph and contains ideas useful for other algorithms for graphs.
- We consider an (undirected) graph $G = (V, E)$ and a distinguished **source** $s \in V$. The algorithm BFS explores all vertex v reachable from s and attributes the **distance** d (i.e. $v.d$), which is the smallest number of edges needed to reach v from s. E.g $s.d = 0$. For any adjacent vertex v of s we have $v.d = 1$, etc.
- The starting point is with all vertices colored with white. BFS visits the vertex and colors them with grey or black.
- If a vertex is discovered first time during the search, then BFS colors the vertex with nonwhite.
- If $(u, v) \in E$ and u is black, then v must be nonwhite.

Breadth-first search (optional part)

- A grey vertex may have some white adjacent vertex. Grey vertices are on the frontier between discovered and undiscovered vertices.
- BFS constructs a breadth–first tree. In the first step there is only the source s, which is the root of the tree. Whenever the search find a white vertex v as adjacent vertex of an already discovered vertex u, the algorithm add the vertex v and the edge (u, v) to the tree and color v with grey. In this case we say that u is the **predecessor**, or **parent**, of v.
- A vertex is discovered when its color change from white to grey. A vertex can be discovered only once and so any vertex has at most one parent. Ancestor and descendant are defined as usual.
- The initial procedure of BFS starts with $G = (V, E)$ represented using adjacency lists, where each vertex $u \in G.V \setminus \{s\}$ are colored white, with distance $u.d = \infty$ and parent NIL denoted by $u.\pi = NIL$

Breadth-first search (optional part)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```



Breadth-first search (optional part)

- When a vertex u is encountered the second time, which is selected with the call DEQUEUE, we discover its white adjacent vertices v, we color then each v grey and adapt $v.d$.Then we color u black and it will not be considered anymore.
- The result of BFS may change according the order in which the adjacent vertices are visited. But we can see that the distance d computed will not.
- One can prove that the running time for BFS is $O(|V| + |E|)$.
- For any vertex v, we define the **shortest–path distance** $\delta(s, v)$ of v from the source s as the minimum number of edges needed in any path from s to v (called a **shortest path**).
- BFS correctly computes shortest path distances.

Single-Source Shortest Paths

- We consider a directed graph $G = (V, E)$ and a weight function $\omega\colon E \to \mathbb{R}_{\geq 0}$ (i.e. with non negative weights).
- The weight $\omega(p)$ of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is given by $\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$.
- For given $u, v \in V$, we define the **shortest–path weight** $\delta(u, v)$ as

$$\delta(u,v) = \begin{cases} \min\{\omega(p) \mid u \overset{p}{\leadsto} v\} & \text{if there is such a path } p \\ \infty & \text{otherwise} \end{cases}$$

- A **shortest–path** from vertex u to vertex v is defined as any path with $\omega(p) = \delta(u, v)$.
- In the following part we consider the so called **single–source shortest–path problem**: from a distinguished vertex $s \in V$, for each $v \in V$, we want to determine a shortest path from u to v.

Optimal substructure of a shortest path

- Shortest–paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.
- For this reason a Greedy algorithm method (that take optimal choice at each step) can be implemented for solving this shortest–paths problem.
- **Lemma**: Let $G = (V, E)$ be a weighted directed graph. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path between the vertex v_0 and the vertex v_k. Then for any integer i and j with $0 \leq i \leq j \leq k$, the path $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a shortest path from v_i to v_j.
  Proof. If there will be a path q_{ij} from v_i to v_j with smaller length, then the path obtained from p by substituting p_{ij} with q_{ij} would have less weight. This would contradict that p is a shortest path.

Representing shortest paths

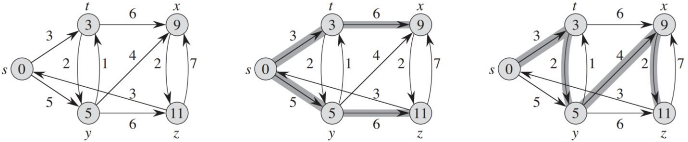
- We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well.
- We define a tree rooted in the source s, so for each $v \in V$ we have the **predecessor** $v.\pi$, that is the parent of v in the tree or it is NIL.
- We have a **predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ where

$$V_\pi = \{v \in V \mid v.\pi \neq \mathrm{NIL}\} \cup \{s\}$$
$$E_\pi = \{(v.\pi, v) \in E \mid v \in V_\pi \setminus \{s\}\}.$$

- In Dikstra's algorithm the subgraph G_π will be a "shortes–paths tree", that is a tree rooted at s, which contains a shortest path from s to each vertex v reachable from s.

A **shortest-paths tree** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \subset V$ and $E' \subset E$, such that

1. V' is the set of vertices reachable from s in G.
2. G' forms a rooted tree with root s.
3. For all $v \in V'$ there is a unique simple path (i.e. with no cycles) in G' from s to v, which is shortest path from s to v in G.



(a) (b) (c)

(b) and (c) provide two examples of shortest-paths tree of the weighted directed graph in (a) with source s.

Relaxation

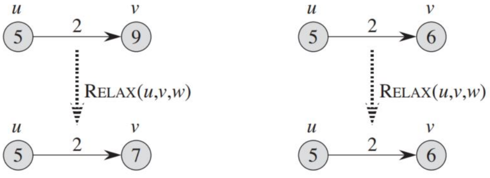
- An important call in Dijkstra's algorithm is RELAX, which applies the technique **relaxation**.
- For a vertex $v \in V$, the attribute $v.d$ (called **shortest–path estimate**) is an upper bound on the weight of a shortest path from the source s and v.
- Therefore the initial step of many shortest–paths algorithms is the following procedure

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

After INIZIALIZATION we have $v.\pi = \mathrm{NIL}$ for all $v \in V$, $v.d = \infty$ all $v \in V$ with $v \neq s$ and $s.d = 0$

Relaxation process takes as input an edge $(u, v) \in E$ and the weight w of the edge (u, v) and tests whether we can improve a "shortest–path" to v found so far by going thorough u. If so it update the attribute $v.d$ (with the smaller value) and $v.\pi = u$.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```



(a) (b)

In (a) after relaxation we have $v.\pi = u$ and $v.d = 7$. In (b) we have before relaxation $v.d = 6 \leq u.d + \omega(u, v)$, thus RELAX does not change $v.d$ and $v.\pi$.

Properties of shortest paths and relaxation

Some of the followings properties can be useful in the analysis of the correctness of shortest-paths algorithms.

- **Triangle inequality**. For all $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + \omega(u, v)$.
- **Upper–bound property**. For all $v \in V$, $v.d \geq \delta(s, v)$, and when $v.d$ achieves the value $\delta(s, v)$ it never changes afterwards.
- **No–path property**. If there is no path from s to v, then $v.p = \delta(s, v) = \infty$.
- **Convergence property**. If $s \rightsquigarrow u \rightarrow v$ is a shortest path for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior the relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

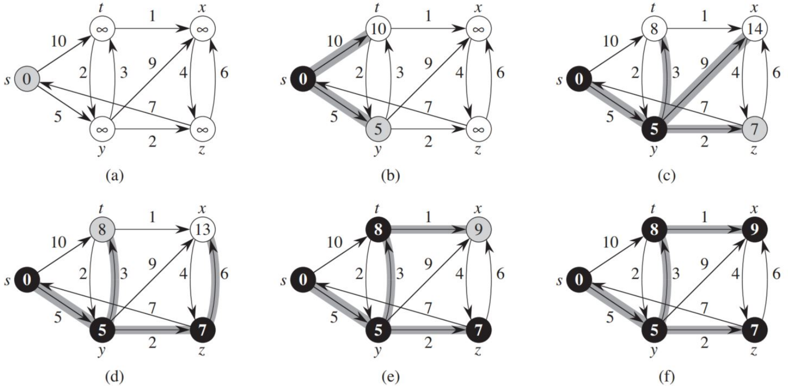
- **Path-relaxation property**. If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.
- **Predecessor-subgraph property**. Once $v.d = \delta(s, v)$; for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Dijkstra's algorithm

- We consider a weighted directed graph $G = (V, E)$, where each weight is non-negative.
- Dijkstra's algorithm starts with INITIALIZE–SINGLE–SOURCE(G,s), so we start with $v.\pi = \mathrm{NIL}$ for all $v \in V$, $v.d = \infty$ all $v \in V$ with $v \neq s$ and $s.d = 0$.
- Dijkstra's algorithm maintains a set S of vertices whose final shortest–path weights from the source s have already been determined.
- The start is with $S = \emptyset$.
- The algorithm repeatedly selects the vertex $u \in V \setminus S$ with the shortest path estimate, adds u to S and relaxes all edges leaving u.
- For a set Q of vertex, EXTRACT–MIN(Q) return a vertex u of Q with the smallest key and give as new set $Q \setminus \{u\}$ which replaces the previous Q.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

The shortest–path estimates appears within the vertices. Shaded edges indicates predecessor values. Black vertices are in S. White vertices are in the min-priority queue $Q = V \setminus S$. The vertex in grey is the EXTRACT–MIN(Q).



- Dijkstra's algorithm with the call EXTRACT–MIN(Q) chooses at step (loop) the light grey vertex, which is the closest vertex in $V \setminus S$ to add to set S. For this reason we said that Dijkstra's algorithm is a Greedy algorithm.
- In general a Greedy algorithm is a "local" optimal algorithm but, it does not always obtain an optimal global/final result.
- One can prove the following:
  Theorem. Let $G = (V, E)$ be a non–negative weighted directed graph. Dijkstra's algorithm runs on G and terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$. The predecessor subgraph G_π is a shortest–paths tree rooted at s.

Reference

The material of these slides is taken form the book "Introduction to Algorithms" by de Cormen et al., Section 22.1, Introduction to Chapter 24 and Section 24.3.