Advanced Algorithms for Data Science
Homework 1

Krikun Gosha

2016-10-07

1 Depth-first and Bread-first search

Theorem 1.1. *In undirected connected graph $G = (V, E)$, if DFS and BFS produce same predecessor subgraph G_π, then $G = G_\pi$ ($V = V_\pi$, $E = E_\pi$).*

Proof. Firstly lets consider BFS (**?**, page 595) and DFS (**?**, page 604) algorithms.

```
    BFS(G, s)
1   foreach vertex u ∈ G.V − {s} do
2       u.color = WHITE
3       u.d = ∞
4       u.π = nil
5   s.color = GRAY
6   s.d = 0
7   s.π = nil
8   Q = ∅
9   ENQUEUE (Q, s)
10  while Q ≠ ∅ do
11      u = DEQUEUE (Q)
12      foreach v ∈ G.Adj[u] do
13          if v.color == WHITE then
14              v.color = GRAY
15              v.d = v.d + 1
16              v.π = u
17              ENQUEUE (Q, v)
18      v.color = BLACK
```

```
    DFS(G, s)
1   foreach u ∈ G.V do
2       u.color = WHITE
3       u.π = nil
4   time = 0
5   foreach u ∈ G.V do
6       if u.color == WHITE then
7           DFS-VISIT(G,U)

    DFS-VISIT(G, u)
1   time = time + 1
2   u.d = time
3   u.color =GRAY
4   foreach v ∈ G.Adj[u] do
5       if v.color == WHITE then
6           v.π = u
7           DFS-VISIT(G,V)
8   u.color =BLACK
9   time = time + 1
10  u.f = time
```

BFS produce subgraph, corresponding to the π (predecessor) attributes.
More formally, for a graph $G = (V, E)$ with source s, we define the **predecessor subgraph** of a bread-first search as

$G_\pi = (V_\pi, E_\pi)$, where
$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup s$
and
$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$

As soon as graph G is connected(given), thus predecessor subgraph of bread-first search forms a **bread-first tree** for G. (by Lemma 22.6 **?**, page 601).

In this case $V_\pi = V$ (bread-first search reach all vertices of graph G).

Let $E_{\text{BFS}} = E_\pi$ in bread-first search.

Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources. Therefore, we define the **predecessor subgraph** of a depth-first search slightly differently from that of a breadth-first search: we let

$G_\pi = (V, E_\pi)$, where
$E_\pi = \{(v.\pi, v) : v \in V_\pi \text{ and } v_\pi \neq \text{NIL}\}$

The predecessor subgraph of depth-first search forms a **depth-first forest** comprising several **depth-first trees**. But as soon as graph G is connected(given), thus predecessor subgraph forms exactly one **depth-first tree**.

Let $E_{\text{DFS}} = E_\pi$ in depth-first search.

By depth-first search we could classify the edges of the connected undirected graph $G = (V, E)$. (Classification of edges **?**, page 609). In undirected graph **forward** and **cross** edges never occurs. (by Theorem 22.10 **?**, page 610).

Predecessor subgraph of bread-first search and predecessor subgraph of depth-first search could be different only in one case, if:

$\exists v \in V : e_{\text{BFS}} \neq e_{\text{DFS}}$, where
$e_{\text{BFS}} \in E_{\text{BFS}}, e_{\text{BFS}} = (v.\pi, v), v.\pi$ - predecessor of v in breadth-first search
$e_{\text{DFS}} \in E_{\text{DFS}}, e_{\text{DFS}} = (v.\pi, v), v.\pi$ - predecessor of v in depth-first search

This situation could be only if in graph G occurs **back** edges. But as soon as BFS and DFS produce same trees, thus all edges in E_{DFS} is **tree** edges and presented in both subgraphs $E_{\text{BFS}} = E_{\text{DFS}} = E$. Therefore graph $G = G_\pi$. □

Remark (Self-loop)**.** *Except for one important remark. This is true for simple graphs. But in general, undirected connected graph could have self-loops. This edges from vertex to itself not present neither in the BFS nor in the DFS.*

2 Minimum spanning tree

Theorem 2.1. *If in weighted undirected graph $G = (V, E, w)$, exists a path between p and q consisting entirely of edges whose cost is smaller than cost of edge $e = (p, q) \in E$, then e does not belong to a minimum spanning tree.*

Proof. Firstly lets consider generic method, which grows the minimal spanning tree one edge in a time. The generic method manages a set of edges A, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we determine an edge (u, v) that we can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a **safe edge** for A, since we can add it safely to A while maintaining the invariant.

```
    GENERIC-MST(G, w)
1   A = ∅
2   while A does not form a spanning tree do
3       find an edge (u, v) that is safe for A
4       A = A ∪ {(u, v)}
5   return A
```

We use cut-and-paste method to determine safe edges and grow the spanning tree (**?**, Theorem 23.1).

Let T be a minimum spanning tree that includes A. Consider cut $(S, V - S)$, such that p and q on the opposite sides of the cut. As soon as graph G has simple path ρ from p to q, $\rho \neq (p, q)$, entirely consisting of edges with less cost (given). Thus cut cross some edge $e \in \rho$ with less cost than (p, q), which will be added in minimum spanning tree T, hereby on each iteration to T added edges from ρ. So path $\rho \in A$ and adding edge (p, q) will produce a cycle, what contradicts minimum spanning tree definition. Therefore edge $(p, q) \notin T$. □

Theorem 2.2. *If in weighted undirected graph $G = (V, E, w)$, edge $e = (p, q) \in E$, does not belong to a minimum spanning tree, then there exists a path between p and q consisting entirely of edges whose cost is smaller or equal cost of e.*

Proof. Let T be a minimum spanning tree that does not include edge (p, q). Consider arbitrary cut $(S, V-S)$, such that p and q on the opposite sides of the cut. Since (p, q) does not belong to a minimum spanning tree, thus cut cross some edge $e \in T$, such what, $w(e) \leq w(p, q)$. Therefore $\exists \rho = \{e_1, e_2 \dots e_n\} \cup \rho \neq (p, q)$, such that $\forall e \in \rho : w(e) \leq w(p, q)$. □

Remark. *Theorem 2.2 claims that path ρ could consists of edges with less or equal costs. There is a case in which (p, q) does not belong to a minimal spanning tree, but simple path does not consists of edges which have less cost strictly, ie $\exists e \in \rho : w(e) = w(p, q)$ but in this ties edge (p, q) does not chosen.*

Corollary 2.3. *Using the above facts, we can propose an $O(m + n)$ algorithm that checks if a given edge e belongs to at least one minimum spanning tree.*

We are interested in finding the simple path ρ from p to q, not equal $\{(p, q)\}$ and consisting entirely from edges with strictly less weight (in case of equal, edge (p, q) could be presented in some MST). If this path exists, edge (p, q) couldn't belongs to any minimum spanning tree by Theorem 2.1.

For finding this path we could use bread-first search with some restrictions on weight of the edge between adjacent vertices and not considering of edge (p, q):

```
     CHECK-MST(G, w, e)
 1   p = e.head
 2   q = e.tail
 3   foreach u ∈ V − {p} do
 4       u.color = WHITE
 5       u.π = nil
 6   Q = ∅
 7   p.color = GRAY
 8   p.π = NIL
 9   ENQUEUE (Q, p)
10   while Q ≠ ∅ do
11       u = DEQUEUE (Q)
12       foreach v ∈ G.Adj[u] do
13           if (v.color == WHITE) ∧ (w(u, v) < w(p, q)) ∧ ((u, v) ≠ (p, q)) then
14               v.color = GRAY
15               v.π = u
16               if v == q then
17                   return doesn't belong to any MST    // ∃ρ = {e_1, e_2 ... e_n} : ∀e ∈ ρ, w(e) < w(p, q)
18               ENQUEUE (Q, v)
19       v.color = BLACK
20   return edge belongs to some MST
```

Consider algorithm complexity - while loop iterates maximum $|V| - 1$ times (minus one for head p of edge). Nested loop "for each" iterates maximum $|E| - 1$ times (one for (p, q) edge). So total complexity $O(|V| + |E|)$.

3 Network flow

Theorem 3.1. *For flow network $G = (V, E)$, in which each edge $(u, v) \in E$ has non-negative integer capacity $c(u, v) \geq 0$, maximum flow is an integer.*

Proof. By Max-flow min-cut theorem (Theorem 26.6 **?**, page 723)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G, where

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

As soon as integer set is closed under the operation of addition, thus $c(S, T)$ is integer and maximum flow in a flow network G is integer as well. □

Theorem 3.2. *For flow network $G = (V, E)$, in which each edge $(u, v) \in E$ has non-negative integer capacity $c(u, v) \geq 0$, if $C = \sum\{c(s, q)|q \in V\}$, then the Ford-Fulkerson algorithm runs in time $O(C \cdot m)$, where m is the number of edges.*

Proof. (Analysis of Ford-Fulkerson algorithm **?**, page 725)

Consider Ford-Fulkerson algorithm:

```
    FORD-FULKERSON(G, s, t)
1   foreach edge(u, v) ∈ G.E do
2       (u, v).f = 0
3   while there exists a path ρ from s to t in residual network G_f do
4       c_f(ρ) = min{c_f(u, v) : (u, v)isinρ} foreach edge(u, v) ∈ ρ do
5           if (u, v) ∈ E then
6               (u, v).f = (u, v).f + c_f(ρ)
7           else
8               (v, u).f = (v, u).f − c_f(ρ)
9   return edge belongs to some MST
```

As soon as C represent bound of maximum flow in cut (S, T), where $S = \{s\}$, thus maximum flow less or equal C. Therefore a straightforward implementation executes the **while** loop of lines 3-8 at most C times, since the flow value increases by at least one unit in each iteration. We can perform the work done within the **while** loop efficiently if we implement the flow network $G = (V, E)$ with the right data structure and find an augmenting path by a linear-time algorithm. Let us assume that we keep a data structure corresponding to a directed graph $G' = (V, E')$, where $E' = \{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$. Edges in the network G are also edges in G', and therefore we can easily maintain capacities and flows in this data structure. Given a flow f on G, the edges in the residual network G f consist of all edges G_f such that $c_f(u, v) > 0$, where c_f conforms to equation:

$$c_f = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

The time to find a path in a residual network is therefore $O(|V| + |E'|)$ if we use either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes $O(|E|)$ time, as does the initialization in lines 1–2, making the total running time of the FORD-FULKERSON algorithm $O(|E| \cdot C)$ □

Exercise 3.2 Phones and base-stations
Naive algorithm for solving this problem - for each base-station $i \in K$ calculate distances to phone $j \in N$, then push to max-heap of station i by distance (except those what are further than max-distance D).
The worst case - $O(K \cdot N \cdot \log N)$.
Then, for each station i take one phone j from top of heap and mark phones as connected. If phone was already connected just take next one (in the worst case this gives another multiplication by K in complexity). Repeat until load-factor of stations reaches the limit (L iterations). Done.
The worst case - $O(K^2 \cdot L \cdot \log N)$.
But this algorithm doesn't connect all possible clients.

Another naive algorithm for solving this problem - for each phone/station construct reachable matrix with total count of reachable station.

	s_1	s_2	...	s_n	total
p_1	0	1	...	1	2
p_2	0	1	...	0	1
...	...	...	...	...	...
p_n	1	0	...	0	k

Time complexity $O(N \cdot K)$, space complexity $O(N \cdot K)$.

Then construct order min-heap by count of reachable stations.
Time complexity $O(N \log N)$, space complexity $O(N)$.

Then extract phone and connect to not fully load station.
Time complexity $O(N \log N \cdot K)$ (another multiplication from search and check of available station in array)

This one should provide a better result.