Fundamental Algorithms

Lecture #11

Cluj-Napoca

December, 11, 2019

Agenda

- **Graphs**
  - **BFS - conclusions**
  - **DSF**
  - **Single Source Shortest Path Dijkstra**

Graphs - **BFS**

- Partition of the vertex set:
  - Unvisited - white (all in the beginning; white->grey at EnQueue)
  - Under visitation - grey; forms the boundary; are all in the Queue
  - Visited - black (all in the end; grey -> black at DeQueue)
- Builds a BFS tree
  - By storing the parent
    - static representation,
    - NOT a dynamic one; π is an array)

$G_\pi (V_\pi, E_\pi)$

$V_\pi=\{v \mid v \in V, \pi[v] <> nil\} \cup \{s\}$

$E_\pi=\{(v, \pi[v]) \mid (v, \pi[v]) \in E, v \in V_\pi-\{s\}\}$

  - It is a forest of trees (in case the graph is not connected)
  - The algorithm needs to restart from every connected component (wrapper over BFS)
  - Runs linearly in the size of the graph (O(|V|+|E|)

BFS – the algorithm

```
bfs (G, s)
for each u ∈V(G) - {s}                         //initialization step
   do color [u] <- white
         d[u] <- ∞                             //distance from source
         π[u] <- nil                           //parent node
color[s] <- grey                               //initialize source
d[s] <- 0
π[s] <- nil
Q <-{s}                                        //initialize Queue (FIFO policy)
while Q <> ∅                         //as long as still have discovered vertices
   do u <- head [Q]          //u – first from the Q; was NOT removed from Q
         for each v ∈ Adj[u]                      //take all the neighbor vertices
                do if color[v] = white  //only from outside the frontier
                      then   color[v] <- grey
                             d[v] <- d[u] +1
                             π[v] <- u
                             EnQ (Q, v)
         DeQ(Q)
color[u]<-black
```

Graphs - **DFS**

- Similar to BFS, the queue is replaced by a **stack**
- (or FIFO policy by LIFO policy) – yet the recursive version more powerful – to discuss
- Keeps the same **color representation** (with similar meaning of colors)
- They (the color clusters) define a vertical boundary between visited/unvisited vertices
- Partition of the vertex set:
  - Unvisited - white (all in the beginning; white->grey at Push)
  - Under visitation - grey; forms the boundary; are all in the Stack
  - Visited - black (all in the end; grey -> black at Pop)

Graphs – **DFS** – additional data

- Keeps the same π - parent node => produces a tree as output, **df-tree** with S as root (also with static representation)
- Add 2 new items representing **time stamps**:

  d[u] = discovery time

  = moment when dfs **first reaches** the vertex u

  f[u] = finishing time

  = moment when dfs **last releases** the vertex u, after visiting ALL its (unvisited) neighbors
- Both types (π and d/f) informative. Lots of properties

DFS – the algorithm

dfs (G, s)

```
for each u ∈V(G)                          //initialization step
   do  color [u] <- white
       π[u] <- nil                        //parent node
time <- 0                   //global variable; keeps track of time evol.
for each vertex u ∈ V[G]                  //take all the neighbor vertices
   do if color[u] = white                 //take only unvisited nodes
       then   dfs_visit(u)
```

dfs_visit (u)

```
color[u] <- grey                          //say u is under visiting process
d[u] <- time                              //just discovered
time<- time+1
for each v ∈ Adj[u]                       //take all the neighbor vertices
   do if color[v] = white                 //only unvisited
       then   π[v] <- u
              dfs_visit(v)                //rec call
color[v] <- black                         //release the node
f[u] <- time
time<- time+1
```

DFS - analysis

- Initialization steps: $O(V)$
- The adjacency list of each vertex is scanned only once (when the vertex reaches the top of the stack): $O(E)$
- The dfs alg: $O(V+E)$ (means $O(|V|+|E|)$)
- The predecessor subgraph ($\pi[s]$) forms a tree /forest of trees (what is forest? When tree/forest?)

$G_\pi(V_\pi, E_\pi)$

$V_\pi=\{v \mid v \in V, \pi[v] <> nil\} \cup \{s\}$

$E_\pi=\{(v, \pi[v]) \mid (v, \pi[v]) \in E, v \in V_\pi-\{s\}\}$

DFS – parenthesis theorem

Th: $G(V, E)$, $\forall$ $u, v \in V$

In any dfs of a directed or undirected graph, time intervals of u, v are:

(1) either entirely disjoint

$$d[u] < f[u] < d[v] < f[v]$$

(2) or one is entirely contained in the other one

$$d[u] < d[v] < f[v] < f[u]$$

v descendant of u

v discovered while u was grey

DFS – edges characterization

- Edge characterization – basis for many other processes (ex. Topological sort, cycles detection, …)
- Edges (u, v) visited in the dfs
  => u always grey (already on the stack)
  v various colors
- How could we create 4 categories out of 3 colors?

- **Tree edges**
- **Back edges**
- **Forward edges**
- **Cross edges**

DFS – edges characterization

- **Tree edges**
  - u = grey; **v = white**;
  - d[u] < d[v] < f[v] < f[u] (intervals included)
- **Back edges**
  - u = grey; **v = grey**
  - d[v] < d[u] < f[u] < f[v] (**u inner v**)
  - link the same branch of the tree backwards =>
  - **They produce cycle**
- **Forward edges**
  - u = grey; **v = black**
  - d[u] < d[v] < f[v] < f[u] (**v inner u**)
  - link the same branch of the tree forward =>
  - Does NOT produce cycles

DFS – edges characterization

- **Forward edges**
  - u = grey; **v = black**
  - d[u] < d[v] < f[v] < f[u] (**v inner u**)
  - link the same branch of the tree forward =>
  - Does NOT produce cycle
- **Cross edges**
  - u = grey; **v = black** (i.e. visit of v finished)
  - d[v] < f[v] < d[u] < f[u] (external times: **() ()**)
  - Does NOT produce cycles
  - Link (Q: how can we differentiate among them?)
    - Different branches of the same tree
    - Different trees of the dfs forest (Q: what's the forest?)

DFS – Benefits

- Produces the *tree structure* (or forest of trees; what's this? Oral discussion) as output in linear time (consider further discussions /seminars for reaching a linear structure as output also in linear time)
- *Time stamps* – have the ability to identify other properties
- *Edge characterization* (many benefits; the immediate one = cycle detection)
  - Cycle detection how? In directed graphs? In undirected graphs? Oral discussion.
- *Parentheses properties*
- DFS may be used as:
  - Skeleton for other algorithms
  - Preprocessing step for other strategies

Single Source Shortest Path the problem - Dijksta

- $G = (V, E)$, $w(u, v)$, $\forall$ $u, v \in V$ $w: E \to R$
- Define a path from source (v_0) to **all** other vertices in the graph: $v_0 \to v_k$
  - Path: $p = < v_0, v_1, \ldots, v_k >$
  - The weight of the path: $w(p) = \Sigma\ w(v_{i-1}, v_i)$, $i=1,k$
  - For each p from v_0, find min w(p), denoted δ
- Notation:

$$\delta(u, v) = \begin{cases} \min \{w(p): \exists\ p\ u \to v\} \\ \infty \qquad \sim\exists\ p\ u \to v \end{cases}$$

- Shortest path from u to v: $w(p) = \delta(u, v)$

Single Source Shortest Path technique

- Greedy
  - Does NOT guarantee optimality (known fact). Same as in MST.
  - Often used for optimization problems (with additional, specific constraints added + proof)
  - Here, the **optimal structure** template
  - Remember Optimal Structure

SSSP- technique Optimal Structure discussion

Dynamic programming - Based on the **optimal structure**

- Reusing instead of recomputing (it pays space to gain time)
- Optimization problems
- Solution to the problem – composition of solutions (of identical problems) on subdomains
- Is it divide et impera? Why not? Oral explanation
- Structure of the problem (Optimal structure): a problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems
- Solution optimal **=>** solution to subproblems **are** optimal (proof by contradiction)
- solution to subproblems optimal **=>** Solution optimal **?**
- NO! WHY? (counterexample)

Notes on **=>**

- Conditional statement (p=>q) is equivalent to its contrapositive (~q=>~p)
- Conditional statement (p=>q) is **NOT** equivalent to its inverse (q=>p)
- (p=>q) ⇔ (~q=>~p)

BUT

- (p=>q) ~⇔ (q=>p)
- Check with truth tables!

SSSP- technique
Optimal Structure discussion

- Dynamic programming - Based on the **optimal structure**
    - Solution to the problem – composition of solutions (of identical problems) on subdomains
    - Solution optimal **=>** solution to subproblems **are** optimal

SSSP
data and functions used

- d[v] = distance from source to v
- π[v] = parent of v on the shortest path from source to v

```
initialize_single_source (G, s)
  for each v∈ V(G)
      do  d[v] <- ∞
          π[v] <- nil
  d[s]<-0
```

SSSP
Relaxation – the technique

- Process that tries to *improve* the shortest path found so far
- Done by taking into account along the path a **new node u**, not yet considered
- Update (**relax**) the path (with parent), if necessary (blackboard)
- Update occurs **only** in case
  - the cost of the path to v so far (d[v]) is **>**
  - than the cost of path to u and (u,v)'weight: d[u] + w(u, v)

SSSP
Relax – the function

```
Relax (u, v, w) //relaxes the cost of the path
        //from source to v by considering the new vertex u
if d[v]>d[u] +w(u, v)
  then d[v]<-d[u] +w(u, v)
       π[v]<-u
```

Update the shortest path from source to v

- if former path better, leave it as it is
- else, in case the new vertex (u) makes any improvement, take the route containing u

Note: a similar techniques was used by Prim's

SSSP Dijkstra's strategy (more than a simple algorithm)

Dijkstra (G, w, s) //1959

```
initialize_single_source (G, s)
S<-∅
Q<-V[G]                //distances updated to edges, where exist
while Q<> ∅
  do u<- extract_min(Q)
     S -> S U {u}
     for each vertex v ∈ Adj [u]
              do relax (u, v, w)
```

Single Source Shortest Path Dijkstra's efficiency

- The original (initial version, in 59) alg. does not use a priority Q, just a regular Q; Analysis for priority Qs
- Q = linear array $O(V^2)$
  - Initial step O(V)
  - Extract min O(V), V times, so $O(V^2)$
  - Each u is added once to S; each edge is inspected once => for loop takes O(E)
- Q = binary heap O(ElgV)
  - Extract min O(lgV), V times, so O(VlgV)
  - Build the initial heap (initial Q) O(V)
  - Relax = decrease key takes O(lgV) applied once to each edge, so O(ElgV)
- Q = Fibonacci heap O(VlgV+E)
  - homework

Single Source Shortest Path Dijkstra's – trace + proof

- Trace – blackboard
- Proof of correctness
  - Informal discussions
    - Relies on the property that subpaths of shortest path are shortest paths as well (optimal substructure)
    - Total Correctness: termination of the algorithm (empty queue) + Partial correctness with Induction on $\delta(u, v)$.
  - Check the textbook (for the formal approach)