Fundamental Algorithms

Lecture #10

Cluj-Napoca

December, 4, 2018

Agenda

- B trees – topic postponed for the end of the course (last lecture or so)
- Graphs
  - Representations
  - Concepts
  - BFS&DSF – just brief intro
  - Topological Sort (seminar)
  - Euler Cycle (Tour) (seminar)
  - Hamiltonian Circuit (seminar – why not)
  - Minimum Spanning Tree (MST) Kruskal
  - Minimum Spanning Tree (MST) Prim
  - BFS

Graphs

- Representation
  - G= (V,E) V = vertex set; E=edges set; E $\subseteq$ VxV
  - adjacency matrix
  - adjacency lists (preferred for sparse graphs)
- Notions
  - directed / undirected graphs / (**dag**)
  - weighted/ unweighted graphs
  - degree of a vertex /in&out degree
  - complete graph
  - diameter of a graph
  - connected / not connected

DFS & BFS importance

- Valuable as stand alone efficient *strategies* (solutions for many real world problems rely on them)
- Valuable for the various *outcomes* they produce
- Useful as *preprocessing* step in other algorithms
- Their *skeleton* useful for the design of other efficient algorithms (by changing /adapting / adding some steps)
- The search could be designed similar, and JUST replace the queue policy
- BFS uses a proper queue (FIFO)
- DFS uses stack (LIFO)

Minimum Spanning Tree (MST) the problem

- $G = (V, E)$, $w(u, v)$, $\forall\ u, v \in V$ //given a weighted graph
- Find $G' = (V, T)$ //find the subgraph
  - $T \subseteq E$, $|T| = |V| - 1$ T is a tree //all nodes, acyclic
  - $\Sigma w(T) = \min$ //weights the fewest
- For finding the MST a greedy strategy is applied, yet an optimal solution is found
  - What is the issue?
  - Discussion (review with highlights on essential aspects) on greedy strategy - orally

Minimum Spanning Tree applications

- Network (real-world problems) design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - TSP (traveling salesperson problem), Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  (method of transmitting a message over a noisy transmission channel; is constructed using a sparse *bipartite graph - seminar*)
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network

Kruskal's MST - approach

- Try to form the tree by joining different partially built trees
- Initially each vertex is in an individual (different) tree (that is, **all vertices** are **in** the solution, **no edge** belong to the solution yet)
- Each step joins the **closest** trees (i.e. 2 trees which are far apart by the smallest edge not yet considered from E) => **selects |V|-1 smallest edges**
- Join should be performed on **different** trees (i.e. do NOT join different branches of the **same** tree => avoid cycle; no longer a tree) – use disjoint sets
  - Unify (x, y) joins 2 disjoint sets – to which x and y belong
  - Find-Set (x) Returns a pointer to the representative element of the set to which x belongs

Kruskal's MST – algorithm

```
MST_Kruskal (G, w)                       //1956
A<-∅        //set A contains the solution in terms of
            // edges considered for addition
for each vertex v ∈ V[G]
  do Build-Set (v)
sort (E) //edges considered for inclusion in order
for each edge (u, v) ∈E  (taken ordered)
  do if Find-Set (u) <> Find-Set (v)
            then A<- A U {(u, v)}
                 Unify (u, v)
return A
```

Kruskal's MST – efficiency; Q&A

- The most expensive step ?
- If optimal (ElgE)
- Trace the algorithm (blackboard; visualize the **tree**)
- Q&A (As orally)
  - Is the solution unique? Justify!
  - Is the algorithm deterministic? Explain!
  - How can we obtain a different solution?
  - All solutions?
  - Prove the Kruskal's solution IS optimal (although applies a greedy strategy – informal justification. Check textbook for a formal and complete proof)

Prim's MST - approach

- 1957 Prim (following Jarnik's 1930 solution)
- Also a greedy approach
- Initially solution **empty** (no vertex, no edge)
- Starts by choosing an initial (ANY) vertex (the root of the tree)
- In each of the **|V|-1** steps, selects the **closest vertex** not yet considered to be added to the tree (add to the solution the selected vertex and the edge connecting it to the so far built tree)
- The structure remains always a tree (i.e. no cycles)
- Keeps 2 additional info:
  - **parent** node (π) - initially all nil
  - **distance** to the parent node (key) – initially all infinity

Prim's MST - algorithm

```
MST_Prim (G, w, r)                 //r = chosen root
Q<-V[G]
for each u∈Q
   do key[u]<- ∞                   //distance to the tree
      π[u] <-nil                   //parent in the tree
key[r]<-0                          //r is the root, hence distance 0
π[r] <-nil                         //root has no parent
while Q<>∅
   do u<-Extract_Min(Q)   //take closest node to the tree
      for each v ∈ Adj[u]
           do if v ∈Q and w(u, v)<key[v]
//for nodes still in Q, in case their distance to the tree
// is larger than the direct edge, update the distance to the tree
                then π[v] <- u
                     key[v]<-w(u, v)
```

Prim's MST – efficiency; Q&As

- Eff. Depends on how the Q is implemented
- If optimal
  - binary heaps
    - Each extract takes lgV; Make E extracts
    - O(ElgV)]
  - Fibonacci heap
    - O(E+VlgV)
- Trace the algorithm (blackboard)
- Q&A (As orally)
  - Advantage over Kruskal?
    - Builds a **rooted tree (π)** – hence you got a tree representation as output
  - Solution unique?
  - Algorithm deterministic?
  - Strategies for finding alternative solutions?
  - Prove the solution obtained by Prim is optimal

Graphs – Fundamental operations Search

- BFS – the basis for MANY important algorithms on graphs (ex: Dijkstra's SSSP; Prim's MST)
- Description – S = source of **BFS**
  - produces a **bf-tree** with S as root
  - bf-tree contains all reachable (from S) vertices of G
  - name – due to the way the frontier is expanded
  - all nodes at distance k from S are discovered before ANY node at distance k+1
  - provides an **exhaustive search** (i.e. if a node is reachable, bfs will eventually find it) yet resource consuming (memory)
  - Q why/where is the memory consumption?
    - keeps paths of all branches

Graphs - **BFS**

- To keep track of the evolution, maintains each category of vertices (unvisited, under visiting, visited) under different sets (using colors attached to vertices)
- **White** – before processing vertices (unvisited)
  - initially all vertices are white;
  - a vertex is in the white set before the bfs reaches the vertex.
- **Gray** – under processing vertices (under visiting)
  - at the time a vertex is discovered, turns to gray;
  - it stays gray as long as it is in the processing stage (the bfs reached the vertex, the visit started yet didn't finish its processing);
  - the set of gray vertices represents the frontier (between processed/unprocessed vertices).
- **Black** – after processing vertices (visited)
  - a vertex turns black when we finished its processing;
  - a vertex is in the black set after the bfs processed the vertex and its (unvisited yet = gray) neighbors

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
```

BFS - analysis

- Initialization steps: O(V)
- The adjacency list of each vertex is scanned only once (when the vertex reaches the head of the Q): O(E)
- The bfs alg: O(V+E) (means O(|V|+|E|)
- The predecessor subgraph (π[s]) forms a tree

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

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

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