Part VI Graph algorithms

Chapter 22 Elementary Graph Algorithms

Representations of graphs

breadth-first-search (BFS)

depth-first-search (DFS)

applications:

(1) topological sort

(2) strongly connected components

representations of graphs

adjacency-list

adjacency-matrix

incidence matrix for directed graphs (The incidence matrix of a directed graph D is a p q matrix $[b_{ij}]$ where p and q are the number of vertices and edges respectively, such that b[i,j]=1 if the edge x_j leaves vertex v_i, 1 if it enters vertex v_i and 0 otherwise. (Note that many authors use the opposite sign convention.)

examples

BFS

The idea of a breadth first search:

”closest nodes” are visited first

data structure to use: *queue*

example:

```
BFS(G, s)

for each u in V[G] - s
  do visit[u] <-- ’unvisited’
     parent[u] <-- null
visit[s] <-- ’visited’
parent[s] <-- null
Q <-- MakeEmptyQueue()
Enqueue(Q, s)
while Not IsEmptyQueue(Q)
  do u <-- Dequeue(Q)
     for each v in Adj[u]
        do if visit[v] = ’unvisited’
             then visit[v] <-- ’visited’
                  parent[v] <-- u
                  Enqueue(Q, v)
return parent
```

BFS tree

cost of BFS $O(|V| + |E|)$

BFS can find a shortest path from s to all other nodes (without weight). But *why?*

DFS

The idea of a depth first search:

"deepest nodes" are visited first

data structure to use: *stack*

```
DFS(G, s)
for each u in V[G] - s
  do visit[u] <-- 'unvisited'
     parent[u] <-- null
visit[s] <-- 'visited'
S <-- MakeEmptyStack()
Push(S, s)
while Not IsEmptyStack(S)
  do u <-- Pop(S)
     visit[u] = 'visited'
     for each v in Adj[u]
        if visit[v] = 'unvisited'
          then parent[v] <-- u
               Push(S, v)
return parent
```

a recursive DFS algorithm (which also generates timestamps)

```
DFS(G)
for each node u in V[G]
 do parent[u] <-- null
time <-- 0
for each node u in V[G]
 do if visit[u] = 'unvisited'
       then  DFS-VISIT(u)
```

```
DFS-VISIT(u)
time <-- time + 1
discover[u] <-- time
visit[u] <-- 'visited'
for each v in Adj[u]
 do if visit[v] = 'unvisited'
      then parent[v] <-- u
           DFS-VISIT(v)
time <-- time + 1
finish[u] <-- time
```

running time ?

Properties of depth-first-search

(1) `u = parent[v]` iff `DFS-VISIT(v)` is called

(2) **parenthesis structure**: for any u, v exactly one of the following three conditions holds:

(a) [`discover[u], finish[u]`] and [`discover[v], finish[v]`] are entirely disjoint, and neither `u` nor `v` is a descendant of the other in the search tree.

(b) [`discover[u], finish[u]` is contained entirely within [`discover[v], finish[v]`] and `u` is a descendant of `v`, OR

(c) [`discover[v], finish[v]` is contained entired within [`discover[u], finish[u]`] and `v` is a descendant of `u`.

WHITE-PATH THEOREM: `v` is a descendant of `u` if and only at time `discover[u]` that the search discovers `u`, node `v` can be reached from `u` along a path consisting entirely of white ('unvisited') nodes.

Classification of edges (for **directed graphs**)

(1) tree edges: those in the search tree (forest)

(2) back edges: those connecting a vertex to an ancester

(3) forward edges: those connecting a vertex to a descendant

(4) cross edges: all other edges

THEOREM 22.10 In a depth-first-search of an **undirected** graph G, every edge of G is either a tree edge or a back edge.

Topological sorting

DAG: directed acyclic graphs

example: edges $\subseteq$ R(prerequisite, course)

reverse order of their finish time

Strongly connected components

Let $G = (V, E)$ be a di-graph. A *strongly connected component* is a maximal subgraph $H = (V_H, E_H)$ of G such that for every two nodes $v, u \in V_H$, there is a path consisting of edges in E_H from v to u and there is a path consisting of edges in E_H from u to v.

Algorithm

```
STRONLY-CONNECTED-COMPONENTS(G)
1. call DFS(G) to compute finish[u] for each u in V[G]
2. compute GT = transpose of G
3. call DFS(GT) (in which vertices are considered
                 in order of decreasing finish[u]
                 as computed in step 1.)
4. output the vertices of each tree in the
   depth-first forest produced by step 3.
```

Properties:

(1) Component graph: $G^{SCC} = (V^{SCC}, E^{SCC})$. G^{SCC} is a dag.

Let C be a SCC, define `finish(C)` $= \max_{u \in C}$ `finish[u]`.

(2) LEMMA 22.14: Let C and C' be distinct strongly connected components for G. If $(u, v) \in E$, where $u \in C$ and $v \in C'$, then $f(C) > f(C')$.

(3) THEOREM 22.16: `STRONLY-CONNECTED-COMPONENTS(G)` correctly computes the strongly connected components for a directed graph G.

Others

Algorithm for computing connected components in undirected graphs.

Reachability problem: given $G = (V, E)$, and $u, v \in V$, is there a path from u to v?

[Is there an SQL program that can solve Reachability problem?]

Chapter 23. Minimum Spanning Trees

spanning trees (MST)

MST: given a connected, undirected graph $G = (V, E)$ with $w : E \to R$, find a spanning tree T such that

$$W(T) = \sum_{(u,v)\in T} w(u, v) \text{ is the minimum}$$

Two greedy algorithms: (1) Kruskal's and (2) Prim's based on a generic MST algorithm.

> The idea: growing an MST by adding one edge to `A` at a time until `A` forms a spanning tree.

But which edge to add??

Growing an MST

```
GENERIC-MST(G,w)

1. A <-- empty
2. while A does not form a spanning tree
3.   do find an edge (u, v) that is safe for A
4.      A <-- A U {(u, v)}
5. return A
```

loop invariant: A is a subset of some MST

safe edge: one does not cause a cycle while maintaining the invariant cut: $(S, V - S)$ is a partition of V

crossing: (u, v) crosses cut $(S, V - S)$ if u and v are in S and $V - S$ respectively (or if v and u are in S and $V - S$ respectively)

respecting: a cut respects a set A of edges if no edge in A crosses the cut.

light edge: an edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

Theorem 23.1 Let $G = (V, E)$ Let `A` be a subset of E that is included in some MST for G, let $(S, V - S)$ be any cut of G that respect A, and let (u, v) be a light edge crossing the cut. Then edge (u, v) is safe for A.

Proof: (1) does not form a cycle; (2) A is still a subset of some MST

Kruskal's algorithm for MST

```
MST-KRUSKAL(G,w)

A <-- empty
for each vertex v in V[G]
  do  MAKE-SET(v)
sort E into nondecreasing order by weight w
for each edge (u, v) in E, taken in order
  do if FIND-SET(u) <> FIND-SET(v)
       then A <-- A U {(u, v)}
            UNION(u, v)
return A
```

disjoint-set data structure and operations:

MAKE-SET, FIND-SET and UNION

running time: $O(|E| \log |V|)$

Prim's algorithm for MST

```
MST-PTIM(G, w, r)

for each u in V[G]
  do key[u] <-- infinite
     parent[u] <-- NULL
key[r] <-- 0
Q <-- V[G]
while Q is not empty
 do u <-- EXTRACT-MIN(Q)
    for each v in Adj[u]
     do if v in Q and w(u,v) < key[v]
          then parent[v] <-- u
               key[v] <-- w(u, v)
 return parent
```

Priority queue: Q

running time? $O(|E| \log |V|)$.

```
UNKNOWN(G, w, r)

for each u in V[G]
  do key[u] <-- infinite
     parent[u] <-- NULL
key[r] <-- 0
Q <-- V[G]
while Q is not empty
 do u <-- EXTRACT-MIN(Q)
    for each v in Adj[u]
     do if w(u,v) + key[u] < key[v]
         then parent[v] <-- u
              key[v] <-- w(u, v) + key[u]
return parent
```

Chapter 24. Single-source shortest paths

single source: s

weighted edges: $w(u, v) \in R$

path weight: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

shortest path weight:

$$\delta(u, v) = \min\{w(p) : p \text{ is a path from u to v}\}$$

Single-source shortest paths: from s to each vertex $v \in V$

Single-destination shortest paths: to vertex t from each vertex $v \in V$

Single-pair shortest path: from s to t

All-pairs shortest paths: from s to t for all pairs $s, t \in V$.

Lemma 24.1 (subpaths of shortest paths are shortest paths)

Given a weighted directed graph $G = (V < E)$ with weight function w. Let $p = (v_1, v_2, \cdots, v_k)$ be a shortest path from v_1 to v_k. Then $p_{i,j} = (v_i, \cdots, v_j)$ is a shorest path from v_i to v_j.

negative weights:

cycles: negative weight cycles, positive weight cycles, 0 weight cycles

representing shortest paths: predecessor π

shortest path tree:

Technique: *relaxation*

Let $d[v]$ be an upper bound on the weight of a shortest path from s to v, initialized ∞.

The process of relaxing edge (u, v): improve $d[v]$ so far by going through u, and update $d[v]$ and $\pi[v]$.

Bellman-Ford algorithm

```
BELLMAN-FORD((G, w, s)
1. for each vertex v in V[G]  {initialization}
2.    do  d[v] <-- infinite
3.        pi[v] <-- null
4. d[s] <-- 0

5. for i <-- 1 to |V| -1      { relaxation}
6.  do for each edge (u, v) in E[G]
7.       do if d[v] > d[u] + w(u, v)
8.            then d[v] <-- d[u] + w(u, v)
9.                 pi[v] <-- u

10. for each edge (u, v) in E[G]  {checking negative}
11.   do if d[v] > d[u] + w(u, v) {weight cycle}
12.        then return FALSE
```

```
13. return TRUE
```

running time : $O(|V||E|)$

Lemma 24.2 Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \rightarrow R$ and assume that G contains no negative weight cycles that can be reached from s. Then after $|V| - 1$ iterations of line 5 in the algorithm, `d[v]` $= \delta(s, v)$ for all vertices v that are reachable from s.

Theorem 24.4 Bellman-Ford algorithm is correct on weighted, directed graphs.

(1) Lemma 24.2 shows the correctness on weighted, directed graphs without negative weight cycles.

(2) we need to show, when G contains a negative weight cycle reachable from s, the algorithm returns FALSE

assume the cycle to be $c = (v_0, v_1, \cdots, v_k)$, where $v_0 = v_k$ and

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

Then because $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$\leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

But

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

implying $\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$.

Single-source shorest paths on DAG

```
DAG-SHORTEST PATHS(G, w, s)

1. topologically sort the vertices of G
2. for each vertex v in V[G]  {initialization}
3.    do  d[v] <-- infinite
4.        pi[v] <-- null
5. d[s] <-- 0

6. for each u in V[G], in topologically sorted order
7.   do for each vertex v in Adj[u]
8.         do if d[v] > d[u] + w(u, v)
9.              then d[v] <-- d[u] + w(u, v)
10.                  pi[v] <-- u

11. return d and pi
```

running time: $O(V + E)$

Dijkstra's algorithm On weighted, directed graphs in which each edge has non-negative weight.

```
DIJKSTRA(G, w, s)

for each vertex v in V[G]  {initialization}
   do  d[v] <-- infinite
       pi[v] <-- null
d[s] <-- 0
S <-- empty
Q <-- V[G]

while Q is not empty
 do  u <-- EXTRACT-MIN(Q)
     S <-- S U {u}
     for each vertex v in Adj[u]
       do if d[v] > d[u] + w(u, v)
             then d[v] <-- d[u] + w(u, v)
```

```
13.                 pi[v] <-- u
```

running time: $O((V+E)lgV)$

Correctness of the algorithm: **Theorem 24.6**, proof by the use of following loop invariant for the *while* loop:

$d[v] = \delta(s, v)$ for each $v \in S$.

Can it deal with negative weight edges?