W4231: Analysis of Algorithms

10/26/1999

- Topological Sort
- Shortest Paths

Topological Sort

Given a **directed** graph $G = (V, E)$, a **topological sort** of the vertices is an ordering $v_1, \ldots, v_n$ of the vertices such that for every edge (v_i, v_j) we have $i < j$.

If the graph has a cycle, the problem is unsolvable.

We will show that if the graph has no cycle, then the problem is solvable.

We will give algorithms that find a topological sort for every acyclic graph.

One Algorithm for “Topological Sort”

1. Find a node v with in-degree zero; make v be the first element of the schedule.
2. Delete v and its incident edges from the graph. Schedule recursively the remaining vertices.

Time: $O(n(n+m))$ with careless implementation.

Correctness: ?

The Optimal and Surprising Algorithm

Algorithm:

- Do DFS; schedule the vertices by decreasing values of $f()$. (Latest finish first)

Claim: if the graph is acyclic, the nodes in the list are ordered in the right way.

Analysis

- **Running time:** $O(m+n)$. We can modify DFS-R so that every time we are finished with a vertex we put it on top of an initially empty linked list.
- **Correctness:** by the following two results:
  - G is acyclic $\Leftrightarrow$ there are no back edges in the DFS forest.
    - We only need $\Rightarrow$
    - $\Leftarrow$ is proved using the “white path theorem.”
  - Cross edges and forward edges always go from nodes with higher finish time to nodes with lower finish time.

First Step

Lemma 1. *If G is acyclic then the DFS forest of G has no back edge.*

PROOF: If there is a back edge then there is a cycle.

The analysis works

Theorem 2. *If G is acyclic, the order of discovery in DFS is a good topological sort.*

PROOF: We want to show that if there is an edge (u,v) then $f(u) > f(v)$. When (u,v) is considered:

- v is not gray, otherwise u would be a descendent of v and (u,v) be a back edge.
- If v is white, v becomes a child of u, and $f(u) > f(v)$.
- If v is black, then $f(v) < f(u)$ too.

A Converse to Lemma 1

Lemma 3. *If the DFS forest of G has no back edge then G is acyclic.*

PROOF: If there is a cycle, let v be the first discovered vertex of the cycle, and let u be the predecessor of v in the cycle.

v is discovered before u, and there is a path (made by all white vertices) from v to u. It follows that u is a descendent of v in the DFS tree (this is quite obvious, but we better prove it later).

Then (u,v) is a back edge.

To complete the argument

Theorem 4. *For any two vertices u and v, exactly one of the following cases hold:*

1. *The intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ are disjoint.*
2. *$[d(u), f(u)]$ contains $[d(v), f(v)]$ and v is a descendant of u in the same DFS tree.*
3. *$[d(v), f(v)]$ contains $[d(u), f(u)]$ and u is a descendant of v in the same DFS tree.*

Theorem 5. [White Path Theorem] *If at time $d(u)$ there is a path of white vertices going from u to v (v included) then v will become a descendant of u in the DFS forest.*

PROOF: Suppose not. Then assume that all the other vertices in the $u \to v$ path become a descendant of u, except v. (Otherwise repeat the argument using instead of v the closest element to u in the path that does not become a descendant.) Then let w be the predecessor of v, then

$$d(u) \le d(v) \le f(w) \le f(v)$$

Then the interval $[d(v), f(v)]$ is contained in $[d(u), f(u)]$ and so v is a descendant of u.

Connectivity Problems in Undirected Graphs

The following problems are easily solved with DFS in optimal $O(n+m)$ time in a given undirected graphs $G = (V, E)$:

- Decide whether G is connected.
- Given a vertex $s \in V$, list all the vertices that are connected to s.
- Given vertices $s, t \in V$, decide whether s and t are connected.

Connectivity Problems in Directed Graphs

The following problems for directed graphs $G = (V, E)$ are also solvable in $O(n+m)$ optimal time:

- Decide whether G is strongly connected (non-trivial).
- Given $s \in V$, list all the vertices that are strongly connected to s (non-trivial).
- Given vertices $s, t \in V$, decide whether s and t are strongly connected (easy with two DFS).

Use of Memory

Consider the simplest problem: given undirected graph $G = (V, E)$, and two vertices $s, t \in V$, decide whether s and t are connected.

The simple DFS solution uses optimal linear time, but also $O(n)$ memory.

Is it possible to use much less memory?

Random Walk

Consider following algorithm:

RandomWalk $(G = (V, E)s, t, T)$
begin
  $v := s$; $c = 1$;
  while $v \neq t$ and $c < T$
    randomly choose a vertex v' such that $(v, v') \in E$
    $v := v'$
    c++
  if $v == t$ **return true**
  else return false
end

One moves at random from a node to a random neighbor starting from s.

When we find t this way, we know that there is a path from s to t and we return **true** .

When we have been moving for more than T steps (where T is a parameter given in input) we give up and we say that presumably t is unreachable from s.

Correctness

If s and t are not connected the algorithm always outputs **false** , which is the right answer.

If s and t are connected, a theorem (that we do not have time to prove) states that

The average number of steps that it takes to go from s to t with a random walk is at most $2nm$

Intuitively, if the average number of steps is $2nm$, then setting $T = 4nm$ should suffice to find t with good probability.

Intermezzo: Markov Inequality

Let X be a random variable that takes non-negative values. Let $k > 0$ be any constant. Then $\mathbf{Pr}[X \geq k] \leq \mathbf{E}[X]/k$.

PROOF:
$$\begin{aligned} \mathbf{E}[X] &= \sum_v v\,\mathbf{Pr}[X = v] \\ &\geq \sum_{v \geq k} v\,\mathbf{Pr}[X = v] \\ &\geq \sum_{v \geq k} k\,\mathbf{Pr}[X = v] \\ &\geq k\,\mathbf{Pr}[X \geq k] \end{aligned}$$

Back to the Analysis of RandomWalk

Consider undirected graph $G = (V, E)$ and connected vertices s and t.

Let L be the random variable that tells us the number of steps that it takes for a random walk starting at s to reach t.

We said $\mathbf{E}[L] \leq 2nm$.

It follows $\mathbf{Pr}[L \geq 4nm] \leq 1/2$.

Let us set $T = 4nm$, i.e. consider what happens when we call $RandomWalk(G, s, t, 4 \cdot |V| \cdot |E|)$.

In general
$\mathbf{Pr}[RandomWalk(G, s, t, T) =$ **false** $] = \mathbf{Pr}[L > T]$.

So $RandomWalk(G, s, t, 4 \cdot |V| \cdot |E|)$ will give the correct **true** answer with probability at least 1/2.

Suppose we want to have error probability $1/1,000,000 \approx 2^{-20}$. We can just repeat the RandomWalk algorithm 20 times, and return **true** if and only at least one invocation of RandomWalk() returned **true** .

```
Connected (G = (V,E)s,t)
begin
  for i := 1 to 20
    if RandomWalk(G,s,t,4|V||E|)==true then return true
  return false
end
```


Distance

Say that the **distance** between s and t is the smallest k such that there is a path of length k connecting s to t. (Distance is undefined, or ∞, is s and t are not connected.)

BFS can be modified to find the shortest path between s and every other vertex.

Rationale

Whenever a new (white) vertex is found, it is reached through a shortest path from s.

Will prove later.

We maintain a vector of distances $d[\cdot]$, where $d[u]$ is the distance from s to u.

Initially, $d[s] = 0$ and $d[u] = \infty$ for $u \neq s$.

Inductively, it will always be true that all vertices in the queue have the right entry in the $d[\cdot]$ vector.

When we are looking at the neighbors of u, the white ones will be at distance $d[u] + 1$ from s.

Modified BFS

```
BFS (s, G = (V,E))
  Initialize Q;
  for all u ∈ V do Initialize col(u) := white
  for all u ∈ V do Initialize d[u] := ∞
  col(s) := gray; d[s] := 0
  enqueue (s,Q)
  while Q is not empty
    u := dequeue (Q)
    col(u) := black
    for all v such that (u,v) ∈ E and col(v) = white do
      col(v) := gray; d[v] := d[u] + 1
      enqueue(v,Q)
```

Correctness

Observations:

- The value of $d[v]$ is set once and for all for every visited vertex.
- $d[v] \geq dist(s, v)$ for every v.
- If $v_1, \ldots, v_q$ are the nodes in the queue, then
  - $d[v_1] \leq d[v_2] \leq \cdots \leq d[v_q]$;

Correctness

We prove by induction: for every vertex v in the queue, $d[v] = dist(s, v)$.

The induction is over the number of steps of the algorithm.

When only s is being considered, then the statement is clear.

Suppose the statement is true up to a point when the queue contains $v_1, \ldots, v_q$. Some new white vertex v is discovered that is a neighbor of v_1. We want to prove $dist(s, v) = d[v_1] + 1$.

Clearly there is a path from s to v of length $d[v_1] + 1$: go from s to v_1 in $d[v_1]$ steps and then from v_1 to v in one step.

Suppose that this were not the best path. Then there would be a path going from s to some intermediate vertex w in less than $d[v_1]$ steps, and then w would be adjacent to v.

But then w would be closer to s than v_1, and so it would have been visited before v_1, but then v could not possibly be a white vertex.

Weighted Graphs

The shortest paths problem becomes more interesting when we consider graphs with *weights* associated to edges. Weights can model the cost or the time needed to go from one node to another or to connect them.

The weights are part of the instance. Can be fields in the adjacency lists or entries in the adjacency matrix.

The weight of an edge (u, v) is typically denoted by $w_{u,v}$.

Paths in weighted graphs

In a weighted graph (be it directed or undirected) the *length* of a path $v_1 v_2 \cdots v_k$

is the sum of the weights of the edges that make up the path $\sum_i w_{v_i, v_{i+1}}$.

Then the case of unweighted graphs can be seen as the special case of weighted graphs where all weights are 1.

Shortest Paths

Given a graph $G = (V, E)$ with non-negative weights on the edges, and two points s and t, we want to find the shortest path connecting s to t.

Generalizations:

1. **Single source shortest paths**. Given s, find for every $u \in V - \{s\}$ the shortest path between s and u.
2. **All pairs shortest paths**. Find the shortest path between any to vertices $u, v \in V$.

The fastest known algorithms for one pair also work for the single source problem, so we will consider only it and the all-pairs one. The all-pairs problem will have to wait until we introduce **dynamic programming**.

Single Source — General Idea

For starters, we show how to only compute the distances. General idea:

- The algorithm works in phases.
- At any phase, it maintains a vector $d[\cdot]$ such that $d[u]$ is always an **upper bound** on the distance between s and u.
- It also maintains a subset S of the vertices, with the property that $d[u]$ is the right distance from s to u for all the vertices u of S.

- At the end, S contains all the vertices, and so $d[\cdot]$ contains the right distances.

Beginning

$d[s] = 0$,

$d[v] = \infty$ for $v \neq s$.

S only contains s.

One Step

Find the vertex u of G not in S with the smallest value of $d[u]$. Include u in S.

For all the nodes $v \notin S$ such that there is an edge between u and v, update $d[]$ as follows:

$$d[v] := \min\{d[v], d[u] + w_{u,v}\}$$

Correctness

Induction on the number of iterations of the algorithm. Inductive step:

Consider the vertex v^* in the graph not in S which is closest to s.

The shortest path from s to v^* only uses elements of S (otherwise there would be some closer element to s outside S).

Therefore it is true that

$$dist(s, v^*) = \min_{u \in S, (u,v^*) \in E} \{dist(s, u) + w_{u,v^*}\}$$

But we know:

- For all vertices u of S, $dist(s,u) = d(u)$, by inductive hypothesis.
- For all vertices $v \notin S$,

$$\begin{aligned} dist(s,v) \le d(v) &\le \min_{u \in S, (u,v) \in E} \{d(u) + w_{u,v}\} \\ &= \min_{u \in S, (u,v) \in E} \{dist(u) + w_{u,v}\} \end{aligned}$$

So at the end of the current step, v is included in S and $d(v) = dist(u,v)$.

Constructing the Tree

- We also maintain a partial tree T all the elements of S. T is the tree of shortest paths from s to the elements of S.
- T is represented by means of a vector $P[]$ of predecessors.
- At the end, the tree reaches all the vertices, and $d[\cdot]$ contains the right distances.

Algorithm

```
ShortestPath(s)
  Initialize P[v] := ∅ for all v
  Initialize d[s] := 0 and d[v] = ∞ for v ≠ s
  CreateQueue(Q)
  Insert(v, Q) for all v ∈ V // The queue is keyed by d[]
  while Q is not empty do
    u := DeleteMin(Q)
    for all v such that (u, v) ∈ E do
      if d[v] > d[u] + w_{u,v} then
        d[v] := d[u] + w_{u,v}
        // Previous step requires a DecreseKey operation in the queue
        P[v] := u
```


Running Time

We do n `insert` and `deletemin` on the heap, and m `decrease-key`.

- Running time $O(n^2)$ is possible implementing the queue with an array. This dates back to Dijkstra (1959).
- $O((m+n)\log n)$ with a heap (1964).
- With Fibonacci heaps, total time is $O(m + n\log n)$. This is optimum for this algorithm in the comparison model [Fredman and Tarjan (1987)].

- If weights are integers that fit into one RAM word, then one can use asymptotically better priority queues. $O(m\sqrt{\log n})$, randomized algorithm using *fusion trees* [Fredman and Willard (1993)]. ... Raman (1996) $O(m + n\sqrt{\log n \log\log n})$ deterministic.

  Departing from Dijkstra's algorithm, Thorup (1997) gives a $O(m+n)$ deterministic algorithm.
- If weights are integers less than U, then van Emde Boas (1977) gets $O(m \log\log U)$ ... Raman (1996) randomized algorithm $O(n(\log U)^{1/4} + m)$.

Directed Acyclic Graphs

It is possible to compute single-source shortest paths in a directed acyclic graph in $O(m+n)$ time.

First do topological sort. Initialize $d[s] := 0$, $d[v] = \infty$.

Then: for all the u that are after s in the sort (respecting the order, starting from s); for all the v such that $(u,v) \in E$,

$d[v] := \min\{d[u] + w_{u,v}, d[v]\}$.