Algorithms (IX)

Yu Yu

Shanghai Jiaotong University

Review of the Previous Lecture

Shortest paths in the presence of negative edges

Negative edges

Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v *must pass exclusively through nodes that are closer than v.*

This no longer holds when edge lengths can be negative.

What needs to be changed in order to accommodate this new complication? A crucial invariant of Dijkstra's algorithm is that the `dist` values it maintains are always either *overestimates* or *exactly correct*.

They start off at 1, and the only way they ever change is by updating along an edge:

UPDATE$((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

Update

UPDATE$((u, v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + \ell(u, v)\}$

1. It gives the correct distance to v in the particular case where u is the *second-last* node in the shortest path to v, and $\texttt{dist}(u)$ is correctly set.

2. It will never make $\texttt{dist}(v)$ too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

Let

$$s \to u_1 \to u_2 \to u_3 \to \cdots \to u_k \to k$$

be a shortest path from s to t.
This path can have at most $|V| - 1$ edges (why?). If the sequence of updates performed includes $(s, u_1), (u_1, u_2), \ldots, (u_k, t)$, *in that order* (though not necessarily consecutively), then by 1 the distance to t will be correctly computed.
It doesn't matter matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are *safe*.

Bellman-Ford algorithm

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?
We simply update all the edges, $|V| - 1$ *times*!

SHORTEST-PATHS(G, ℓ, s)
Input: Graph $G = (V, E)$;
edge lengths $\{\ell_e \mid e \in E\}$ with no negative cycles, vertex $s \in V$
Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

```
1. for all u ∈ V do
2.         dist(u) = ∞
3.         prev(u) = nil
4. dist(s) = 0
5. repeat |V| − 1 times:
6.         for all e ∈ E do
7.                 update(e)
```

Running time: $O(|V| \cdot |E|)$.

Negative cycles

If the graph has a *negative cycle*, then it doesn't make sense to even ask about shortest path.

How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$ iterations, perform one extra round.
There is a negative cycle if and only if some dist value is reduced during this final round.

Shortest paths in dags

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

graphs without negative edges, and *graphs without cycles*.

We already know how to efficiently handle the former. We will now see how the single-source shortest-path problem can be solved in just *linear time* on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

In any path of a dag, the vertices appear in increasing linearized order.

A single-source shortest-path algorithm for directed acyclic graphs

DAG-SHORTEST-PATHS(G, ℓ, s)

Input: Dag $G = (V, E)$;
edge lengths $\{\ell_e \mid e \in E\}$, vertex $s \in V$

Output: For all vertices u reachable from s, $\texttt{dist}(u)$ is set to the distance from s to u.

```
1.  for all u ∈ V do
2.          dist(u) = ∞
3.          prev(u) = nil
4.  dist(s) = 0
5.  Linearize G
6.  for each u ∈ V in linearized order do
7.          for all edges (u, v) ∈ E do
8.                  update(e)
```

Notice that our scheme doesn't require edges to be positive.
In particular, we can find longest paths in a dag by the same algorithm: *just negate all edge lengths*.

Greedy algorithms

Minimum spanning trees

Building a network

Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a *maintenance cost*.

The goal is to

- pick enough of these edges that the nodes are *connected*,
- the total maintenance cost is *minimum*.

Properties of the optimal solutions

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called **trees**. A tree with *minimum total weight*, is a **minimum spanning tree**.

Input: An undirected graph $G = (V, E)$; edge weights w_e
Output: A tree $T = (V, E')$ with $E' \subseteq E$ that *minimizes*

$$\text{weight}(T) = \sum_{e \in E'} w_e.$$

Trees

Lemma (2)

A tree on n nodes has $n - 1$ edges.

Lemma (3)

Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

Lemma (4)

An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

A greedy approach

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

The correctness of Kruskal's method follows from a certain *cut property*.

The cut property

Lemma (**Cut property**)

Suppose edges X are part of a minimum spanning tree (MST) of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V \setminus S$, and let e be the lightest edge across this partition. Then

$$X \cup \{e\}$$

is part of some MST.

A **cut** is any partition of the vertices into two groups, S and $V \setminus S$.

The cut property says that it is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V \setminus S$), provided X has no edges across the cut.

Proof of the cut property

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. *This cycle must also have some other edge e' across the cut $(S, V \setminus S)$.*

If we now remove e'

$$T' = T \cup \{e\} \setminus \{e'\}$$

which we will show to be a tree.

T' is connected by Lemma (1), since e' is a cycle edge. And it has the same number of edges as T; so by Lemmas (2) and (3), it is also a tree.

Proof of the cut property (cont'd)

T' is a minimum spanning tree:

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type. Therefore $w(e) \leq w(e')$, and

$$\text{weight}(T') \leq \text{weight}(T).$$

Since T is an MST, it must be the case that $\text{weight}(T') = \text{weight}(T)$ and that T' is also an MST.

Kruskal's algorithm

KRUSKAL(G, w)

Input: A connected undirected graph $G = (V, E)$ with edge weight w_e

Output: A minimum spanning tree defined by the edges X.

```
1. for all u ∈ V do
2.         makeset(u)
3. X = {}
4. Sort the edges E by weight
5. for all edge {u, v} ∈ E in increasing order of weight do
6.         if find(u) ≠ find(v) then
7.                 add edge {u, v} to X
8.                 union(u, v)
```

$\lvert V\rvert$	`makeset`(x)	create a singleton set containing x
$2 \cdot \lvert E\rvert$	`find`(x)	find the set that x belongs to
$\lvert V\rvert - 1$	`union`(x, y)	merge the sets containing x and y

A data structure for disjoint sets

Union by rank

We store a set is by *a directed tree*. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the *root* of the tree.
This root element is a convenient *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a *self-loop*.

In addition to a parent pointer π, each node also has a **rank** that, for the time being, should be interpreted as the height of the subtree hanging from that node.

Union by rank

MAKESET(x)

$\pi(x) = x$
$\texttt{rank}(x) = 0$

FIND(x)

while $x \neq \pi(x)$ **do** $x = \pi(x)$
return x

`makeset` is a constant-time operation.

`find` follows parent pointers to the root of the tree and therefore takes time *proportional to the height of the tree*.

The tree actually gets built via the third operation, `union`, and so we must make sure that this procedure keeps trees *shallow*.

Union

$\underline{\text{UNION}}(x, y)$

$r_x = \texttt{find}(x)$
$r_y = \texttt{find}(y)$
if $r_x = r_y$ **then** return
if $\texttt{rank}(r_x) > \texttt{rank}(r_y)$
    then $\pi(r_y) = r_x$
else
    $\pi(r_x) = r_y$
    if $\texttt{rank}(r_x) = \texttt{rank}(r_y)$ **then** $\texttt{rank}(r_y) = \texttt{rank}(r_y) + 1$

Properties

Lemma (1)

For any x that is not a root, $\mathtt{rank}(x) < \mathtt{rank}(\pi(x))$

Lemma (2)

Any root node of $\mathtt{rank}$ k has least 2^k nodes in its tree.

Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of $\mathtt{rank}$ k.

With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E| \log |V|)$ for sorting the edges,
- $O(|E| \log |V|)$ for the `union` and `find` operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that *sorting can be done in linear time*?

THEN THE DATA STRUCTURE PART BECOMES THE BOTTLENECK!

The main question:

How can we perform union's and find's faster than log n?

Path compression

$\underline{\text{FIND}}(x)$

if $x \neq \pi(x)$ **then** $\pi(x) = \text{FIND}(\pi(x))$
return $\pi(x)$

The benefit of this simple alteration is *long-term* rather than instantaneous and thus necessitates a particular kind of analysis:

we need to look at sequences of `find` and `union` operations, starting from an empty data structure, and determine the *average time per operation*.

This *amortized cost* turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Time analysis

Think of the data structure as having a "top level" consisting of the root nodes, and below it, the insides of the trees.

There is a division of labor:

- `find` operations (with or without path compression) only touch the insides of trees,
- `union`'s only look at the top level.

Thus path compression has no effect on `union` operations and leaves the top level unchanged.

Time analysis (cont'd)

We now know that the ranks of root nodes are unaltered, but what about nonroot nodes?
The key point here is that once a node ceases to be a root, it never resurfaces, and its rank is forever fixed.
Therefore the ranks of all nodes are unchanged by path compression, even though *these numbers can no longer be interpreted as tree heights*.

In particular,

- For any x that is not a root, $\texttt{rank}(x) < \texttt{rank}(\pi(x))$
- Any root node of `rank` k has least 2^k nodes in its tree.
- If there are n elements overall, there can be at most $n/2^k$ nodes of `rank` k.

Time analysis (cont'd)

If there are n elements, their rank values can range from 0 to $\log n$.

Divide the nonzero part of this range into the following intervals:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \ldots, 16\}, \{17, 18, \ldots, 2^{16} = 65536\},$$
$$\{65537, 65538, \ldots, 2^{65536}\}, \ldots$$

Each group is of the form $\{k + 1, k + 2, \ldots, 2^k\}$ where k is a power of 2.

The number of groups is $\log^* n$, which is defined to be the number of successive log operations that need to be applied to n to bring it down to 1 (or below 1).

For instance, $\log^* 1000 = 4$ since $\log \log \log \log 1000 \leq 1$. In practice there will just be the first five of the intervals shown; more are needed only if $n > 2^{65536}$, in other words *never*.

Time analysis (cont'd)

In a sequence of `find` operations, some may take longer than others.
We'll bound the overall running time using some *creative accounting*.

We will give each node a certain amount of **pocket money**, such that the total money doled out is at most $n \log^* n$ dollars.

We will then show that each `find` takes $O(\log^* n)$ steps, plus some additional amount of time that can be paid for using the pocket money of the nodes involved – *one dollar per unit of time*.

Thus the overall time for m `find`'s is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

Time analysis (cont'd)

A node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed.

If this rank lies in the interval $\{k+1, \ldots, 2^k\}$, the node *receives* 2^k *dollars*. By Property 3, the number of nodes with rank $> k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}.$$

Therefore the total money given to nodes in this particular interval is at most n dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $n \log^* n$.

Time analysis (cont'd)

Now, the time taken by a specific `find` is simply the number of pointers followed.

Consider the ascending rank values along this chain of nodes up to the root. Nodes x *on the chain* fall into two categories:

- either the rank of $\pi(x)$ is in a higher interval than the rank of x,
- or else it lies in the same interval.

There are at most $\log^* n$ nodes of the first type, so the work done on them takes $O(\log^* n)$ time.
The remaining nodes – whose parents' ranks are in the same interval as theirs – have to pay a dollar out of their pocket money for their processing time.

Time analysis (cont'd)

This only works if the initial allowance of each node x is enough to cover all of its payments in the sequence of `find` operations.

Here's the crucial observation: each time x pays a dollar, its parent changes to one of *higher rank*.

Therefore, if x's rank lies in the interval $\{k+1, \ldots, 2^k\}$, it has to pay at most 2^k dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.

Set cover

The problem

A county is in its early stages of planning and is deciding *where to put schools*.

There are only two constraints:

- each school should be in a town,
- and no one should have to travel more than 30 miles to reach one of them.

What is the minimum number of schools needed?

This is a typical *set cover* problem. For each town x, let S_x be the set of towns within 30 miles of it. A school at x will essentially "cover" these other towns. The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

Set cover problem

SET COVER

Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$

Output: A selection of the S_i whose union is B.

Cost: Number of sets picked.

This problem lends itself immediately to a greedy solution:

> *Repeat until all elements of B are covered: Pick the set S_i with the largest number of uncovered elements.*

The greedy algorithm doesn't always find the best solution!

Performance ratio

Lemma

Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).
Since these remaining elements are covered by the optimal k sets, there must be some set with at least n_t/k of them. Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right),$$

which by repeated application implies

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t.$$

□