

Note: this set of notes (and those for the next several lectures) are just my own notes that I used in class. We didn't necessarily cover everything on the notes on a particular day, so some material is repeated. Also, all the material is available in the Dasgupta book, chapters 3, 4, and 5, I believe. -jfh

On to BFS: flamefront idea reviewed. What about BFS in a graph with edge-lengths?

Idea for dijkstra:

graph on A, B, C; AB = 100; AC = 200; BC = 50

introduce 100, 200 extra nodes

run BFS. Nothing "interesting" happens for 100 steps.

Set an alarm clock to wake us when that happens.

When we reach node A, it turns out that we might get to B in just 50 more steps, so reset the B alarm for 150 instead of 200.

In general: advance to next alarm; reset all newly-lower alarms.

Code:

```
procedure dijkstra(G; l; s)
```

```
//Input: Graph G = (V,E), directed or undirected;
```

```
//    positive edge lengths {l_e : e in E}; vertex s in V
```

```
//Output: For all vertices u reachable from s, dist(u) is set
```

```
//    to the distance from s to u.
```

```
for all u in V :
```

```
    dist(u) = infty
```

```
    prev(u) = nil
```

```
dist(s) = 0
```

```
H = makequeue(V) // priority queue, using dist-values as keys
```

```
while H is not empty:
```

```
    u = deletemin(H)
```

```
    for all edges (u, v) in E:
```

```
        if dist(v) > dist(u) + l(u, v):
```

```
            dist(v) = dist(u) + l(u, v);
```

```
            prev(v) = u
```

```
            decreasekey(H; v, dist(v)) // to new value of dist(v)!
```

Analysis: building a heap-based PQ: $O(|V|)$

deletemin: $O(\log |V|)$; done $|V|$ times $\rightarrow |V| \log |V|$

decreasekey: done $|E|$ times (maybe). $|E| \log |V|$

$\rightarrow O((|V| + |E|) \log |V|)$

One view of Dijkstra: “greedy”: after k rounds, we know shortest paths to k nodes. To make that $k+1$, we find the “adjacent” node that’s closest, and adjust ITS distance. That’s greedy.

What if some distance is NEGATIVE? Dijkstra can fail. Sigh.
What if some CYCLE is negative?
observation:

update((u, v) in E)
 $\text{dist}(v) = \min(\text{dist}(v), \text{dist}(u) + w(u, v))$

1. Gives correct value to $\text{dist}(v)$ if $\text{dist}(u)$ is correct, and u is second-to-last in path from s to v .
2. It will never make $\text{dist}(v)$ too small, so it’s “safe”.

What if we just update over and over? How many steps in the shortest path from s to v ? $|V|-1$ edges, max. (Why?)

If the path is $s, u_1, u_2, \dots, u_k, t$
and we update $(s, u_1), (u_1, u_2), \dots, (u_k, t)$, IN THAT ORDER, then $\text{dist}(t)$ will be set correctly, regardless of what other updates take place. How to be sure they’re done in that order? DO ALL of them $|V|-1$ times!

proc shortest-paths(G, w, s)
input: directed graph $G = (V, E)$
 edge weights w with no negative cycles
 starting vertex s in V

output: for all verts reachable from s , $\text{dist}(u)$ is distance from u to s .

forall u in V
 $\text{dist}(u) = \text{infty}$
 $\text{prev}(u) = \text{nil}$

$\text{dist}(s) = 0$
repeat $|V|-1$ times
 forall e in E
 update(e)

“Bellman-Ford” algorithm
Have students do Bellman-Ford on graph

Observations:

1. Good to check whether any updates occur; often all shortest-paths are short, so only need a few rounds of updates
2. Go ahead and do it $|V|$ times. If there's an update on the $|V|$ th round, then there's a negative cycle!

New topic: Minimum Weight Spanning Tree

Assumption: CONNECTED graphs. Why? Disconnected case: find MWST for each component. Find components via DFS.

Problem statement:

Input: An undirected graph $G = (V, E)$ with edge weights w_e

Output: a tree, $T = (V, E')$, where E' is a subset of E , minimizing the sum of the weights, and with the result connected.

Tree: a connected acyclic graph.

Observations on trees:

1. Removing an edge disconnects
2. If T has n nodes, it has $n-1$ edges
3. If $G = (V, E)$ is connected and undirected, and $|E| = |V|-1$, then G is a tree
4. An undirected graph is a tree iff there's a unique path between any two nodes.

Observation about MWST: removing a cycle edge cannot disconnect a graph.

Greedy: add next lightest edge that doesn't make a cycle

Problem: need to know whether endpoints are in same component.

Solution: Union/Find. Blindingly efficient when many operations are done. Operations are "makeset", "union", and "find". If there are n items, *find* is $O(\log n)$. We won't prove this. In practice, it's much faster. So Kruskal is $O(|V| \log |V|)$

```
forall u in V
  makeset(u)
```

```
X = {} // the MST
sort edges in E by weight
foreach edge (u,v) in E, in increasing order of weight
  if (find(u) != find(v)):
    add edge (u,v) to X
    union(u, v)
```

```
return X
```

```
proc makeset(x)
```

```
parent(x) = x
rank(x) = 0
```

```
proc find(x)
  while x != parent(x) : x = parent(x)
  return x
```

```
proc union(x, y)
  rx = find(x); ry = find(y); // find parent.
  if rx == ry: return
  if rank(rx) > rank(ry):
    parent(ry) = rx;
  else
    parent(rx) = ry
  if rank(rx) == rank(ry) : // only time rank increases!
    rank(ry) = rank(ry) + 1
```

[Improved find: during traversal, re-point each node to the root! Makes amazing efficiency!]

Alternative: build up MWST by building a tree on k nodes ($k = 0, 1, 2, \dots, |V|$)
At each stage, add the edge from the current set to the untouched set that has min weight.

Jarnik: 1930
Prim: 1957
Dijkstra: 1959

```
proc prim (G, w, u0)
  forall u in V
    cost(u) = infty
    prev(u) = nil
  cose(u0) = 0
  H = makequeue(V, cost()) // priority queue based on costs of edges
  while H not empty
    v = deletemin(H)
    foreach (v, z) in E
      if (cost(z) > w(v, z))
        cost(z) = w(v, z)
        prev(z) = v
```

result: the “prev” array defines the tree.