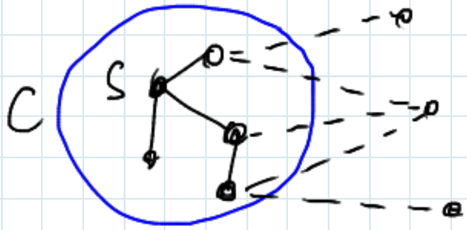
Recall Minimum Spanning Tree Problem.

last day:
Kruskal's Algorithm & implementation $O(m \log n)$

Prim's Algorithm.

Grow one connected component in a greedy fashion (i.e. by adding min. weight edge leaving the component)



Choose min. weight edge leaving C

C = set of vertices reached by T so far

```
initialize $C \leftarrow \{s\}$, $T \leftarrow \emptyset$
while $C \neq V$
    find min. weight edge $e = (u, v)$ from
        $u \in C$ to $v \in V - C$
    $T \leftarrow T \cup \{e\}$
    $C \leftarrow C \cup \{v\}$
end
```

Correctness The exact same exchange argument works. And in fact, we could prove one lemma that gives correctness of both algs. (see text).

Prim – implementation.

In general, we need to find min. weight edge leaving C, the connected component of T.

Priority Queue data structure

Maintain set of weighted elements (in our case, edges leaving C)

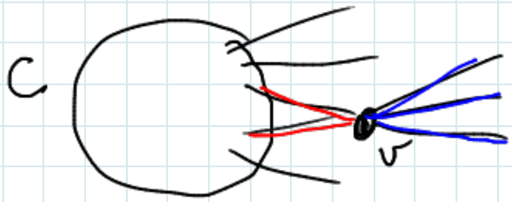
Operations

- Find and delete min weight element
- Insert
- Delete

Can be implemented as a heap (see CS 240 or text) at O(log k) time per operation, k = # elements

In our case

$\delta(C)$ = edges leaving C



Changes to $\delta(C)$ when v is added to C:

- edges from C to v leave $\delta(C)$
- other edges adjacent to v enter $\delta(C)$

We can find these edges by going through v's adjacency list.

Each edge enters $\delta(C)$ once and leaves it once

Priority Queue operations

$n-1$ find min

m insert

m delete

Total cost

$$O(\underset{\text{Find}}{\underset{\uparrow}{n}} \log m + \underset{\text{Insert, Delete}}{\underset{\uparrow}{m}} \log m) = O(m \log m)$$
$$= O(m \log n)$$

It is slightly more efficient to keep a priority queue of vertices $V - C$ with weight(v) = min weight edge from C to v

size of PQ $= n$

update is key-change $O(\log n)$

Still gives $O(m \log n)$ total.

Additional improvement

use Fibonacci heap to implement PQ

Then decrease key is $O(1)$

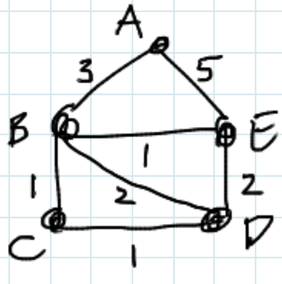
so we get $O(\underbrace{n \log n}_{\text{find}} + \underbrace{m}_{\substack{\text{decrease key} \\ \text{(key change)}}}) = O(m + n \log n)$

Shortest Paths in Edge Weighted Graphs

Recall that BFS from v finds shortest paths from v in unweighted undirected graphs.

General input: directed graph with weights on edges.

Note: Can represent undirected edge as 2 directed edges



Shortest path A – D is ABD, weight 5.
A – E ABE, weight 4.

We will sometimes allow negative weights but we'll assume no negative weight cycle (otherwise go around it ∞ to get −∞ length)

[Note: we might still want a shortest path that is simple (doesn't repeat vertices) but that's NP-complete]

Versions of the problem:

1. Given u, v, find shortest uv path
2. Given u, find shortest uv path ∀ v
   "single source shortest path problem"
3. Find shortest uv path ∀ u, v
   "all pairs shortest path problem"

Solving 1 seems to involve solving 2.
But we can solve 2 faster than 3.
Start with 2. Do 3 later (dynamic programming)

Single Source Shortest Paths in Directed Graphs

- general weights (but no neg. cycle) $O(n \cdot m)$
  Bellman Ford
- no cycles $O(n+m)$
- no negative weights $O(m \log n)$
  Dijkstra's algorithm

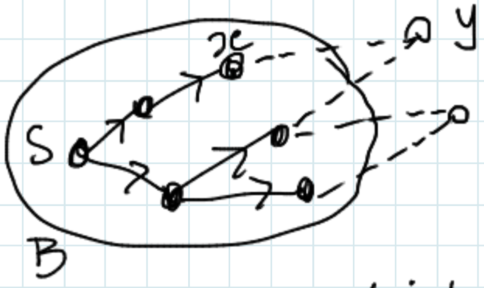
Dijkstra's Algorithm 1959

Input: digraph $G=(V,E)$, $w: E \to \mathbb{R}^{\geq 0}$, $s \in V$

non-neg. edge weights   source

Output: shortest path from s to every other vertex v.

Idea: Grow tree of shortest paths starting from s



General step: have tree of shortest paths to all vertices in set B

initially $B = \{s\}$

Choose edge (x,y) $x \in B$, $y \notin B$
to minimize $d(s,x) + w(x,y)$

distance from s to x — known

Note similarity & differences to Prim's MST alg.

Call this min. d

$d(s,y) \leftarrow d$

add (x,y) to tree ($\text{Parent}(y) \leftarrow x$)

This is greedy in the sense that we always add the vertex with next min distance from s.

Claim d is the min. distance from s to y.
[this justifies the output being a tree]

Proof. Any path π from s to y consists of

π_1 – initial part of path in B
(u,v) – first edge leaving B
π_2 – rest of path.

$$w(\pi) \geq w(\pi_1) + w(u,v) \geq d(s,u) + w(u,v) \geq d$$

using that $w(\pi_2) \geq 0$
– the proof breaks down for neg. weight cycles

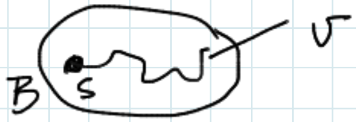
Therefore, by induction on $|B|$, the alg. correctly finds $d(s,v)$ for all v.

Implementations

- want to choose edge leaving B to minimize some value
- could make a heap of edges (x,y) $x \in B$, $y \notin B$
  where $\text{value}(x,y) = d(s,x) + w(x,y)$
  This heap has size $O(m)$
- More efficient: a heap of vertices

Keep "tentative distance" $d(v)$ $\forall v \notin B$

$d(v)$ = min. weight path from s to v with all but last edge in B



```
Initialize
    $d(v) \leftarrow \infty$   $\forall v \neq s$
    $d(s) \leftarrow 0$
    $B \leftarrow \emptyset$
While $|B| < n$
    $y \leftarrow$ vertex of $V \setminus B$ with min. d value  - from heap
    $B \leftarrow B \cup \{y\}$   -- note that $d(y)$ is true distance
    for each edge $(y,z)$ do
        if $d(y) + w(y,z) < d(z)$ then
            $d(z) \leftarrow d(y) + w(y,z)$  - and update heap
            Parent $(z) \leftarrow y$
        end
    end
end
```

Store d values in a heap. size is $\leq n$

Modifying a d value takes $O(\log n)$ to adjust heap

Total time $O(\underset{\substack{\uparrow \\ \text{find min}}}{n} \log n + \underset{\substack{\uparrow \\ \text{adjust heap}}}{m} \log n) = O(m \log n)$

Actually, there is a fancier "Fibonacci heap" that gives $O(n \log n + m)$ see CLRS