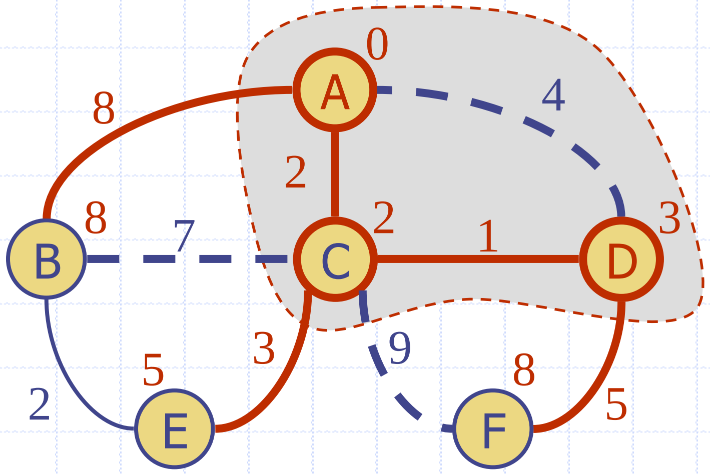
Lesson 14
Algorithms For Weighted Graphs:
Creative Intelligence Manifesting As Material Creation

Wholeness of the Lesson

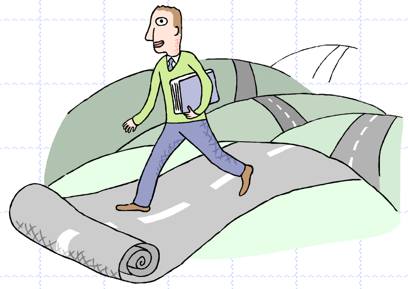
Weighted graphs are graphs that have *weights* or *costs* associated with each edge. Two questions that often need to be answered when working with weighted graphs are (1) What is the least costly path between two given vertices of the graph? (2) What is the least costly subgraph of the given graph which includes all the vertices of the given graph? Dijkstra's Shortest Path Algorithm provides an efficient solution to the first question; Kruskal's Minimum Spanning Tree Algorithm provides an efficient solution to the second. Solutions to optimization problems of all kinds give expression to Nature's tendency to achieve the most possible with the least expenditure of energy. Waking up to one's own deeper values of intelligence has the effect of drawing Nature's style of functioning into our thinking and action so that we automatically achieve goals with less effort.



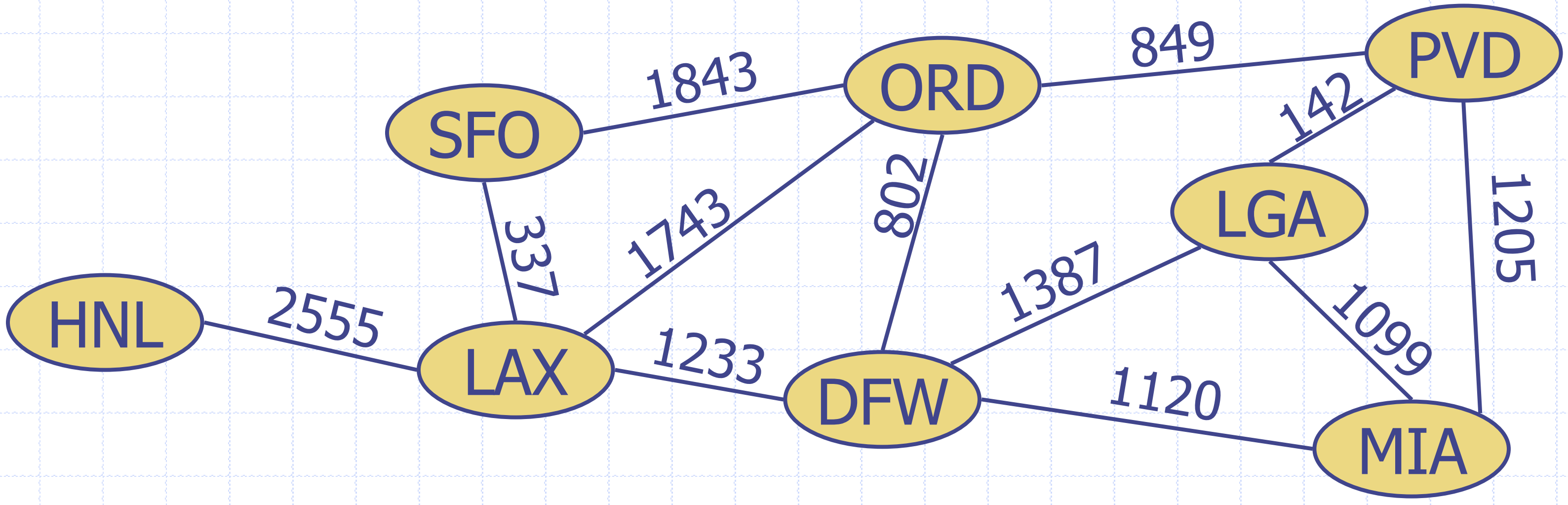
Outline

- Weighted graphs
- Shortest path problem
- Dijkstra’s algorithm
- Minimum spanning tree problem
- Kruskal’s Algorithm

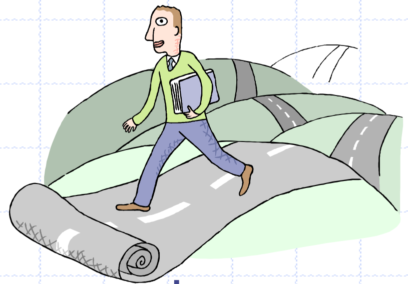
Weighted Graphs



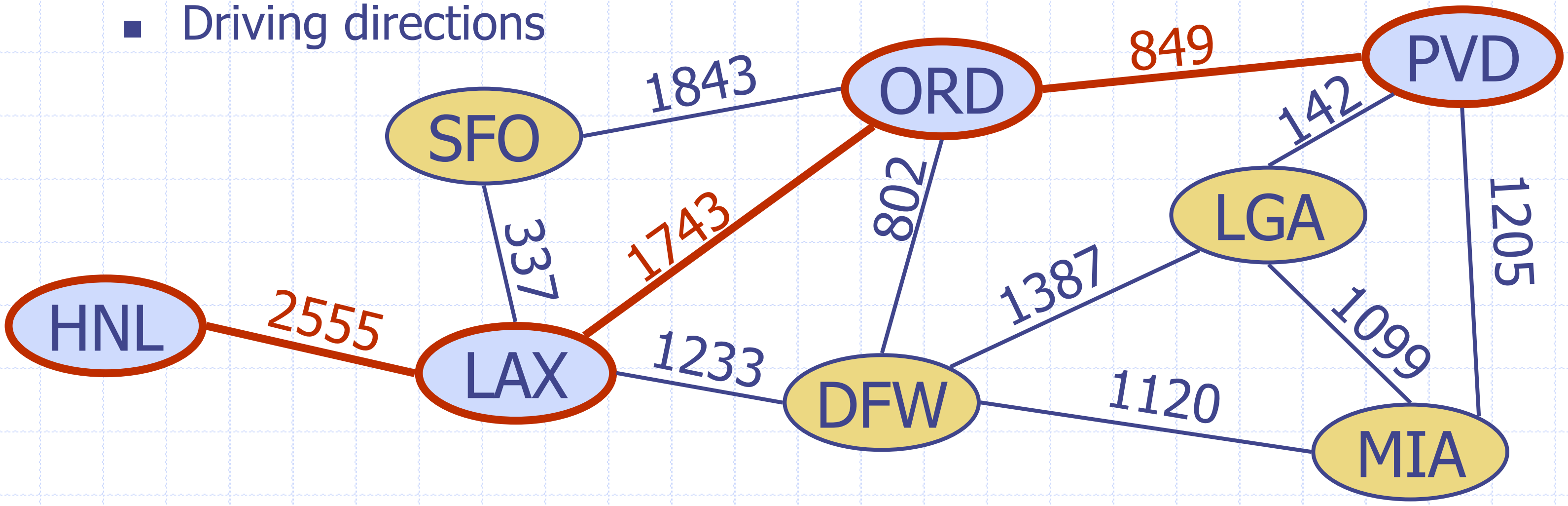
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge (wt: edges → numbers)
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



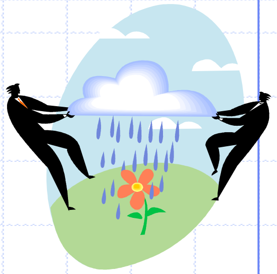
Shortest Path Problem



- Given a connected weighted graph and two vertices s and x, we want to find a path of minimum total weight between s and x.
  - "Length" of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



Dijkstra's Algorithm: The Problem



- The *distance* of a vertex ***v*** from a vertex ***s***, denoted ***d*(*s*,*v*)**, is the length of a shortest path between ***s*** and ***v***
- ***Question:*** Is it always true in a weighted graph that, for any two vertices *v*, *w*, d(*v*,*w*) = wt(*v*,*w*)? Prove or give a counterexample.
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex ***s***
- ***Dijkstra's Algorithm History***
- QuickSort History
- Assumptions:
  - the graph G = (V,E) is connected
  - the edges are undirected
  - the edge weights are positive

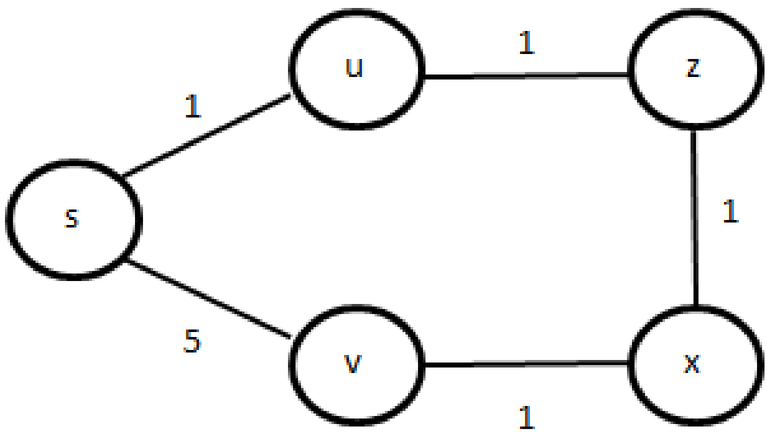
- Starting with weighted graph G = (V,E) and starting vertex s, we wish to compute, for each vertex v, the shortest distance from s to v in G.
- We will store our computed value of the distance from s to any vertex v in an array A:

  A[v] = our computed value of distance from s to v
- If our algorithm is right (which we will need to prove) then for each v in V, A[v] = d(s,v).

The Basic Idea

- *Basic Strategy:* It is reasonable to attempt to compute distances first for vertices close to s, then for vertices farther away.
- Step 1. We set A[s] = 0 (since d(s,s) = 0).
- Step 2. Pick a vertex v adjacent to s so that (s,v) has the least weight among all edges incident to s.
  - It will turn out that for this v, a shortest path from s to v really is wt(s,v). We will set A[v] = wt(s,v).
  - However, it will not be true in general that for any other w adjacent to s, wt(s,w) is a shortest path from s to w

Example



<u>The Logic</u>: Why does Step 2 always work? (Pick a vertex u adjacent to s so that (s,u) has least weight among edges incident to s.)

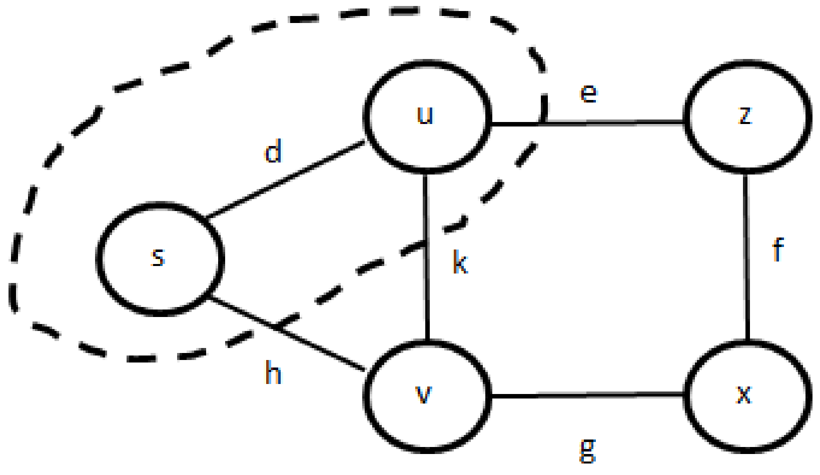
If p : s, w_1, w_2, …, w_k = u is any other path from s to u, then w_1 must itself be u (otherwise the first part of the path – s, w_1 – is already longer than just s, u)

- Setting A[u] = wt(s,u) = 1 is correct
  - (s,u) has least weight among all pairs (s,t) where t adjacent to s
- However, setting A[v] = 5 is not correct
  - A less costly path from s to v than s,v is s, u, z, x, v

Basic Idea, continued

- Step 3. After an optimal vertex v adjacent to s has been chosen, there are two possible ways to extend further
  - For some other vertex w adjacent to s, wt(s,w) may turn out to be the shortest path length from s to w, OR
  - For some w adjacent to v, the path s, v, w is shortest possible from s to w.
  - Among these choices, the algorithm will pick the path that has minimal total weight.
  - Note: The vertex w may be adjacent to s or v.

Example of Step 3



Compute minimum of

wt(s, v)

A[u] + wt(u,z)

A[u] + wt(u,v)

One of these will be a correct minimum path length (either from s to v or from s to z)

Why Step 3 Works

Case I. wt(s,v) is minimum

Any path from s to v begins either as s, u or s, v. By assumption s, v is shorter than s, u, v. The other possibility is s, u, z, x, v, but s, v is (by assumption) already shorter than s, u, z.

Case II. A[u] + wt(u,v) is minimum

Case III. A[u] + wt(u,z) is minimum.

Cases II and III are similar. We prove it for Case III: We want to show that s, u, z is a shortest path to z. The other possible choices are s, u, v, x, z and s, v, x, z

Path s, u, v, x, z:
length s,u,z ≤ length of s,u,v (by assumption)

Path s, v, x, z:
length of s, u, z ≤ length s, v (by assumption)

Dijkstra's Algorithm

- We extend Steps 1, 2, 3 till every vertex has been reached
- In general, we build a subset X of the set V of vertices, beginning with s and eventually including all of V. Associated with each vertex v in X will be a value A[v] representing the algorithm's computation of the shortest path length from s to v
- In each step, we add one new vertex w to X. We use a *greedy strategy* to select w : vertex w will be an endpoint of an edge (v,w) from X to w whose *greedy length* (= A[v] + wt(v,w)) is the least possible among all edges having one end in X, the other outside of X.
- Once a vertex w arrives in X, the storage array A will be updated so that A[w] stores the greedy length found for (v, w). It will be shown that once the value A[w] is computed, it is the correct shortest path length from s to w – namely, we show A[w] = d(s, w).

Dijkstra’s Algorithm

Input: A simple connected undirected weighted graph *G* with nonnegative edge weights, determined by a weight function *wt* (*x,y*), and a starting vertex *s* of *G*.

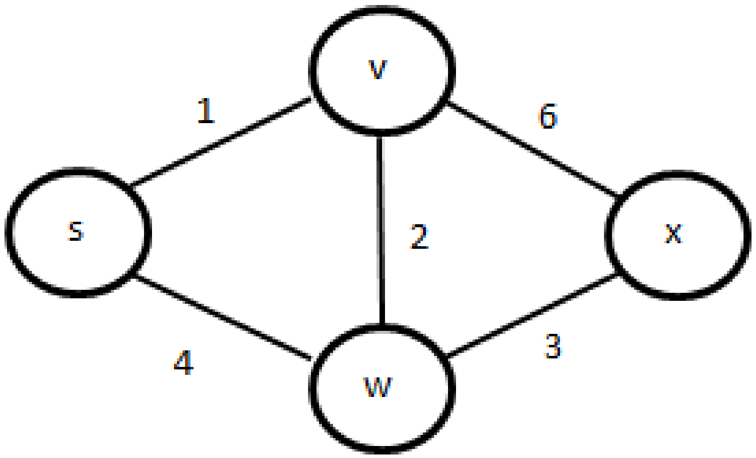
Output: Array A of shortest distances d(s,v) from s to v, for each v in V, so A[v] = d(s,v) for each v
Aux Output: Array B with property that B[v] is a shortest path from s to v.

The Algorithm:

```
A [s] ← 0. B [s] ← empty path (empty set)
X ← {s}  //Basis step

while  X ≠ V do
    { POOL ← {(v,w) ∈  E | v ∈ X and w ∉ X} }
    //here, we have no control over how much of E is searched, leading to slow running time
    (v′,w′) ← search POOL for edge (v,w) for which A[v] + wt(v,w) is minimal
    add w′ to X
    A[w′] ← A[v′] + wt(v′,w′)
    B[w′] ← B[v′] U {(v′,w′)}
```

- Consider an example.

Worked Example: Step 1



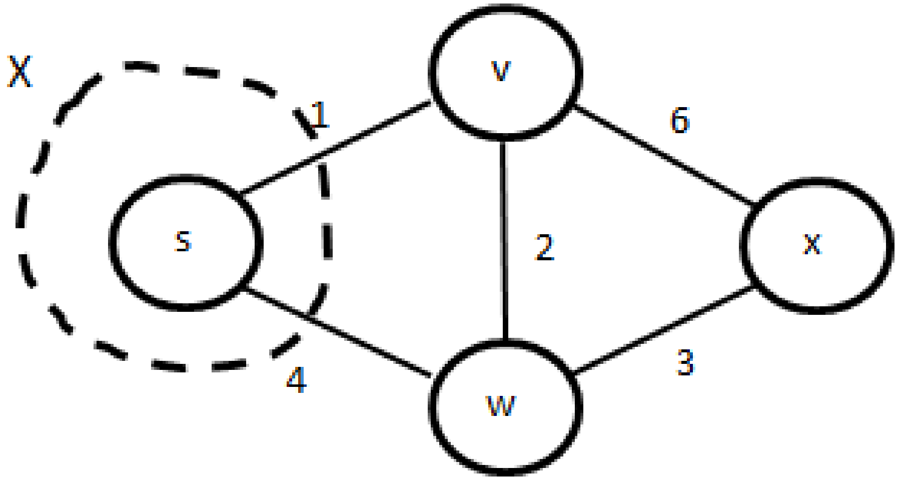
Step 1.

A[s] $\leftarrow$ 0

B[s] $\leftarrow$ { }

Put s in X

Worked Example: Step 2



Step 2.

X = { s }

POOL = { (s,v), (s,w) }

Find minimum greedy length – min of the following

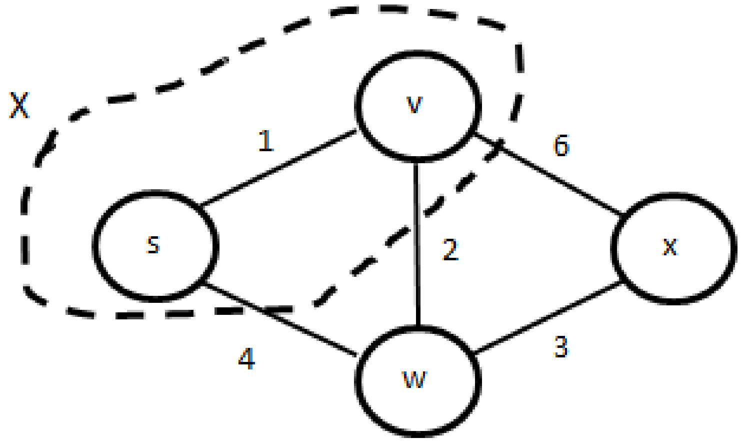
A[s] + wt(s,v) = wt(s,v) = 1 ←

A[s] + wt(s,w) = wt(s,w) = 4

Add v to X and set value of A[v]: A[v] ← 1

Auxiliary Storage: B[v] = B[s] U {(s,v)} = {(s,v)}

Worked Example: Step 3



Step 3.

X = { s, v }

POOL = { (s,w), (v,w), (v,x) }

Find minimum greedy length – min of the following

A[s] + wt(s,w) = wt(s,w) = 4

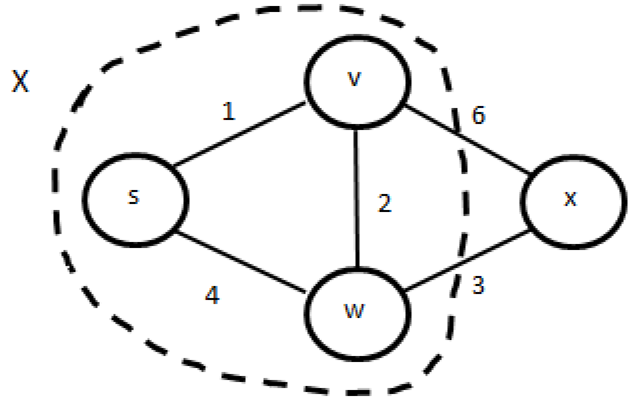
A[v] + wt(v,w) = 1 + wt(v,w) = 1 + 2 = 3 ←

A[v] + wt(v,x) = 1 + wt(v,x) = 1 + 6 = 7

Add w to X and set value of A[w]: A[w] ← 3

Auxiliary Storage: B[w] = B[v] U {(v,w)} = {(s,v), (v,w)}

Worked Example: Step 4



Step 4.

X = { s, v, w }

POOL = { (w, x), (v,x) }

Find minimum greedy length – min of the following

A[v] + wt(v,x) = 1 + wt(v,x) = 1 + 6 = 7

A[w] + wt(w,x) = 3 + wt(w,x) = 3 + 3 = 6 $\leftarrow$

Add x to X and set value of A[x]: A[x] $\leftarrow$ 6

Algorithm complete since X = V. Computed values:

A[s] = 0   A[v] = 1   A[w] = 3   A[x] = 6

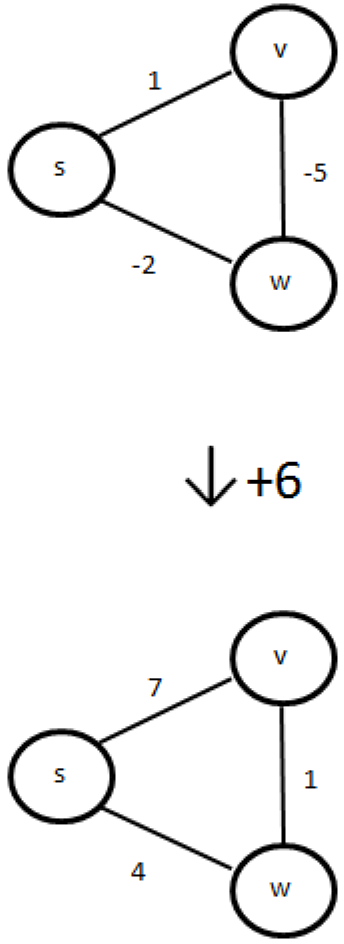
Computeed values of array B:
B[s] = { }, B[v] = {(s,v)}, B[w] = {(s,v), (v,w)}, B[x] = {(s,v), (v,w), (w,x)}

Dijkstra - Exercises

- Why is there a requirement that edges have *non-negative* weights? Why can't we add a large positive constant to every edge (to eliminate negative edge weights) and compute shortest paths for the new graph using Dijkstra?

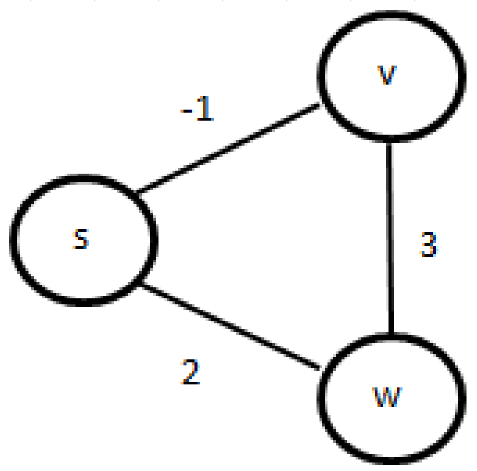
Dijkstra - Exercises

- Why is there a requirement that edges have *non-negative* weights? Why can't we add a large positive constant to every edge (to eliminate negative edge weights) and compute shortest paths for the new graph using Dijkstra?



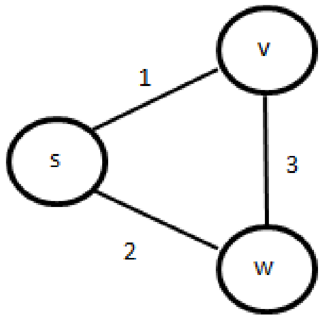
Exercises, continued

Does Dijkstra's Algorithm *sometimes* work correctly when there are negative edge weights? Consider this weighted graph.

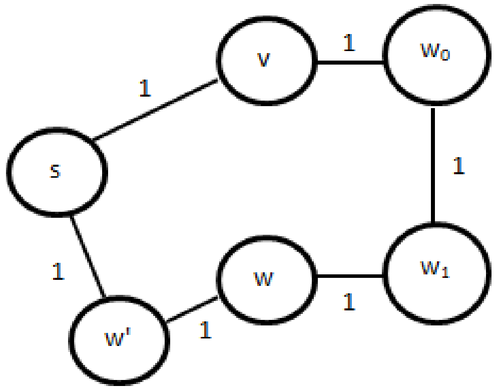


Exercises, continued

Why is Dijkstra's approach to the shortest path problem better than simply using BFS, as described in the previous lesson?



↓BFS Style



[BFS approach: Making all edge weights = 1 is same as removing all weights. Perform BFS with start vertex s and compute distance to each vertex by returning its *level* in the BFS spanning tree. These computed values should be same as values found using Dijkstra]

Dijkstra – Correctness

- Loop Invariant: I(i) is the following statement: (where i means iteration #i)
  - (1) $|X| = i + 1$
  - (2) $A[v] = d(s,v)$ for all $v \in X$
- Preconditions for loop:
  - $X = \{s\}$, $A[s] = 0$ (Basis Step)
- Postconditions for loop:
  - $X = V$, $A[v] = d(s,v)$ for all $v \in V$

Dijkstra – Correctness (2)

Verification of I(i) for all iterations i = 1 … n-1. We verified base case i = 1 in earlier slides.
Induction Step: We assume I(i) is true, so
|X|= i + 1 and A[v] = d(s,v) all v in X.

- Iteration i+1 causes one more vertex to be added to X, so |X| = i + 2
- During iteration i+1, algorithm locates (v′,w′) that has least greedy length among edges from X to V - X, and the algorithm sets A[w′] = A[v′]+d(v′,w′)
- To complete the induction, it suffices to show A[w′] is shortest path length from s to w′,i.e., A[w′] = d(s,w′)

Dijkstra – Correctness (3)

- Let q : s, …, y, z, …, w′ be a truly shortest path from s to w′, where z is first vertex in V - X encountered on the path q. Let L be the length of q. Let q_0 be the path s, …, y, z; we denote its length L_0. Notice that $L_0 \leq L$ (since no edge has negative weight). We will actually show that $A[w'] \leq L_0$, and this will finish the induction step.

- Notice that the sum of edge weights in q_0 from s to y is the true distance d(s,y) from s to y because q is a shortest path from s to w′ (if we could find a shorter path from s to y, we could also find a shorter path from s to w′). Therefore, by the induction hypothesis,

$$L_0 = \text{length of } q_0 = d(s,y) + wt(y,z) = A[y] + wt(y,z).$$

- Recall from the previous slide that the algorithm so far has already defined $A[w'] = A[v'] + wt(v',w')$ and that this is the smallest sum of the form $A[u] + wt(u,w)$, for u in X and w not in X.

- It follows that $A[v'] + wt(v',w') \leq A[y] + wt(y,z)$ and so $A[w'] \leq L_0$. This completes the induction and proof of correctness.

Dijkstra – Running Time

- Running time without optimizations can be computed by observing that a (potentially) exhaustive search of edges is made in each iteration, leading to a running time of O(mn).

- This can be improved to O(m * log n) if an optimal data structure is used.

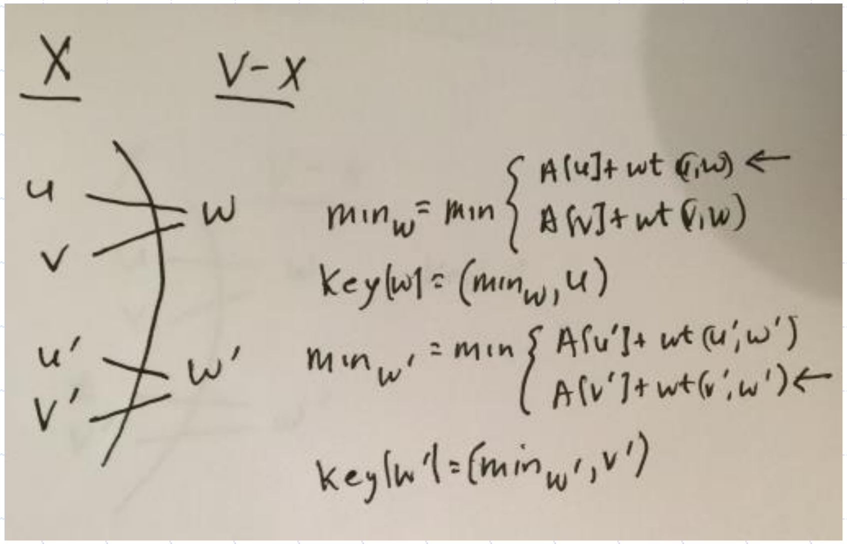
- *Idea*: Since "mins" are needed in each iteration, serve them using a Priority Queue instead of doing an exhaustive search of edges.

Dijkstra – Running Time (2)

- We use a priority queue to store vertices still in V - X. The key key[w] to be used with each vertex w in V - X is defined by

  key[w] = $(\text{min}_{w,}v)$

  where min_w is defined to be the minimum among all sums A[u] + wt(u,w), where u is in X and (u,w) is in E, and v is the value of u for which min_w = A[v] + wt(v,w).


X
V-X
u
v
w
u'
v'
w'
$\text{min}_w = \text{min}\begin{cases} A[u]+wt(u,w) \leftarrow \\ A[v]+wt(v,w) \end{cases}$
$\text{key}[w] = (\text{min}_w, u)$
$\text{min}_{w'} = \text{min}\begin{cases} A[u']+wt(u',w') \\ A[v']+wt(v',w') \leftarrow \end{cases}$
$\text{key}[w'] = (\text{min}_{w'}, v')$

Dijkstra – Running Time (3)

In each iteration, the single step of locating (v′,w′) via an exhaustive search of edges is replaced by two steps:

- `removeMin` to get the pair (key[w′],w′), where key[w′] includes both v′ and the sum A[v′]+wt(v′,w′).
- update the heap

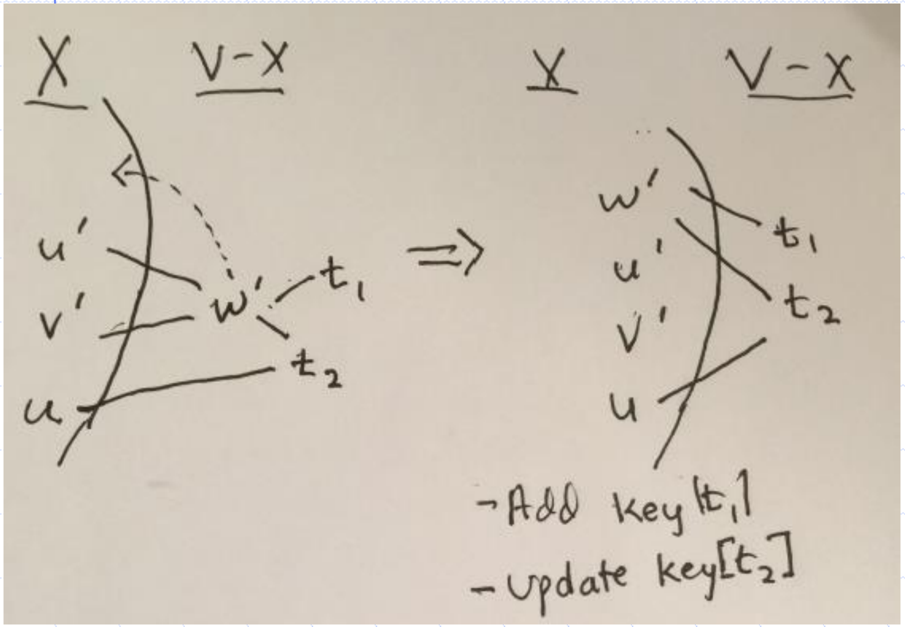
Dijkstra – Running Time (4)

Running time for handling the heap operations

(1) `removeMin` requires Θ(log n) (including downheap to restore heap)

(2) update the heap:

When w′ is moved to X, edges between w′ and other vertices in V - X must be taken into account



a. There may be new keys for t ∈ V – X and (w′,t) in E – for each such t, new key given by:

   key[t] = (A[w′] + wt(w′,t), w′)

   [Running time to insert: O(log n)]

b. There may be a t ∈ V – X that already has a key, but key may need to be updated because of w′ – re-compute min among A[x] + wt(x,t) for which x in X. If update is needed, (key[t], t) must be removed, key[t] set to new value, and (key[t], t) re-inserted

   [Running time: O(log n)]

Dijkstra – Running Time (5)

- Summing over all iterations, running time has the following asymptotic bound [Note: for each v in V: removeMin is O(log n) and updating heap is O(deg(v) * log n)]

$$\sum_{v \in V} (1 + \deg(v)) \log n = (n + 2m) \log n \text{ which is } O((n + m) \log n)$$

- Since the graph is connected, n is O(m).
- Running time is therefore

O(m log n)

(this improves the O(mn) running time of the naïve algorithm)

Improved Dijkstra Correctness

The proof that this optimized version of Dijkstra’s algorithm is also correct is essentially identical to the proof for the naïve algorithm. The only point that needs to be observed is that, at each step of either algorithm, the edge (v’,w’) that is chosen has exactly the same properties whether it is obtained using the naïve or optimized approach; namely, (v’,w’) has the least greedy length among all pairs (v,w) for which v belongs to X and w lies outside of X. The only difference in the algorithms is *how* (v’, w’) is found, but it has the same essential characteristic in either case. And the discussion above demonstrates that the method for choosing (v’, w’) used by the optimized algorithm is correct (and performs better than the method used in the naïve algorithm). We conclude that the optimized Dijkstra algorithm is also correct.

Main Point

Dijkstra's algorithm is an example of a *shortest-path algorithm* – an algorithm that efficiently (O(mlog n)) computes the shortest distance between two vertices in a graph.

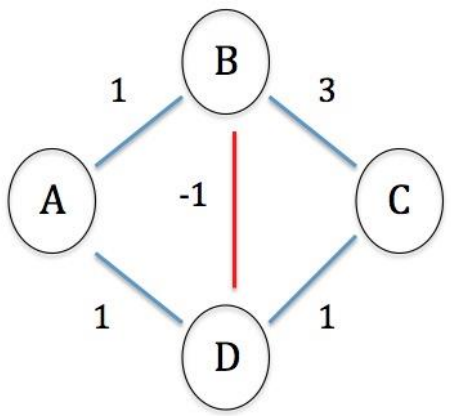
Analogously, Nature itself is known to obey the law of least action – Nature does the least possible amount of work to proceed from one location or state to another. Nature's way of achieving this makes use of computational dynamics that involve "no effort" and no steps.

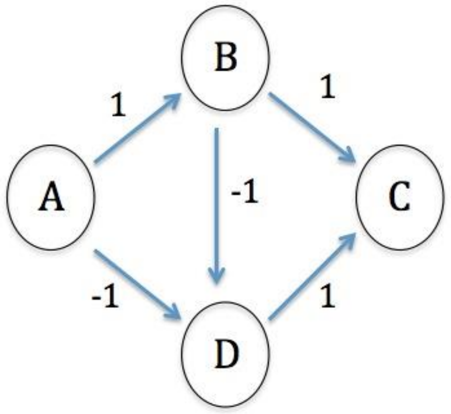
Dijkstra’s Algorithm for Directed Graphs

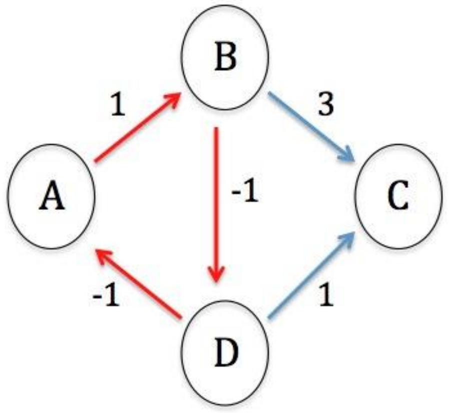
- Essentially the same algorithm can be used to find the shortest path from a starting vertex to any other vertex in a directed graph (as long as it is reachable). That means, all weights must be non-negative. If at least one weight is negative, we must use Dynamic Programming algorithm given in Slide 36.

Computing Shortest Paths with Negative Edge Weights

- Even if there is just one edge with a negative weight in an undirected graph, it is impossible to compute shortest paths between vertices.
- This limitation can be removed by restricting attention to digraphs.







There is no shortest path because of the negative edge weight:
A-B-D-C
A-B-D-B-D-C
A-B-D-B-D-B-D-C

In a directed graph, negative weight edges are not an inherent problem, with one exception . . .

Even in a directed graph, the presence of *negative weight cycles* makes it impossible to compute shortest paths.

A Dynamic Programming Approach for DAGs (Directed Acyclic Graphs)

Given a weighted DAG G = (V,E), a starting vertex s, and terminating vertex t, we observe something about the shortest path length d(s,t) from s to t:

- Let in(t) = {w_1, w_2, . . ., w_k} be in-vertices for t (that is: for each i, (w_i, t) is an edge of G).
- Every path from s to t must include a pair w_i, t at the end, for some i between 1 and k.
- Therefore d(s,t) must be the minimum among all sums d(s,w_i) + wt(w_i, t), for $1 \leq i \leq k$. Therefore, the problem of computing d(s,t) is reduced to computing distances d(s,w_i) involving fewer vertices (subproblems) and combining results by computing a minimum among sums d(s,w_i) + wt(w_i, t).
- This observation leads to a dynamic programming solution.

First Try

For each v, plan to store (memoize) d(s,v) in an array D, so that D[v] = d(s,v)

```
D[s] = 0
for v $\in$ V – {s}
    D[v] = $\infty$
    for (w,v) $\in$ E do
        //want to read D[w] from storage
        if D[w] + wt(w,v) < D[v] then
            D[v] $\leftarrow$ D[w] + wt(w,v)
```

Problem: No guarantee that D[w] has been computed when it is needed in the computation of D[v].

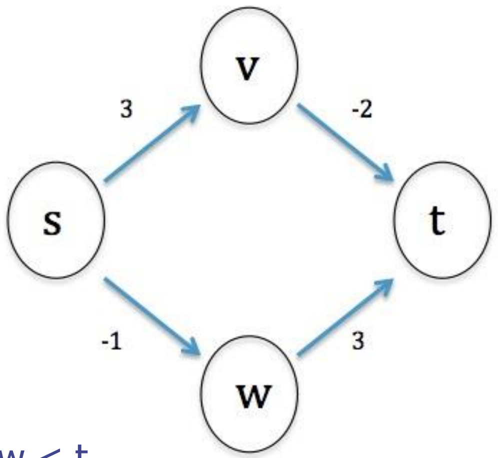
Second Try

A solution is to put vertices in an order that ensures D[w] has already been computed when it is needed during computation of D[v] (where $(w,v) \in E$): Try a *topological sort*, arranging V into $<v_1, v_2, . . ., v_n>$ as a preprocessing step. Then in the algorithm:

```
D[s] = 0
for v ∈ V – {s}
   D[v] = ∞
   for (w,v) ∈ E do
      //we can now read D[w] from storage
      if D[w] + wt(w,v) < D[v] then
            D[v] ← D[w] + wt(w,v)
```

We are now guaranteed that in looping through in-vertices w for v, w has been examined before v in the outer loop, since $w \rightarrow v$ implies w precedes v in the sequence $<v_1, v_2, . . ., v_n>$ (because of the topological ordering of V)

Example of Dynamic Programming Shortest Path Algorithm



Topological ordering of V: s < v < w < t

D[s] = 0
D[v] = min {D[x] + wt(x,v) | (x,v) $\in$ E}
= D[s] + 3 = 3
D[w] = min{D[x] + wt(x,w) | (x,w) $\in$ E}
= D[s] - 1= -1
D[t] = min {D[x] + wt(x,t) | (x,t) $\in$ E}
= min{D[v] + -2, D[w] + 3}
= min {(3 – 2), (-1 + 3)}
= 1

Exercise: What happens if you try to compute the shortest directed path length from s to t using Dijkstra's Algorithm?

Correctness

- Suppose after running the algorithm there is some v for which D[v] is not the shortest length of a directed path from s to v. Assume v is the first in the topological ordering of V for which this failure occurs. Certainly $v \neq s$.

- Let p: $s \rightarrow \ldots \rightarrow w \rightarrow v$ be a shortest path from s to v. Then p′: $s \rightarrow \ldots \rightarrow w$ is a shortest path from s to w, and, since vertices preceding v in the topological ordering have correct D values, D[w] stores the length d(s,w) of p′. Therefore, the shortest path length from s to v is d(s,w) + wt(w,v) = D[w] + wt(w,v).

- When the algorithm executed and v was examined in the outer loop, one of the edges that was considered is (w,v), and the sum D[w] + wt(w,v) was tested as a candidate for D[v]. But since this is the smallest value, it would have been chosen during execution of the inner loop. Contradiction!

Running Time

- Recall that topological sort can be done in $O(n + m)$ time.

- After that preprocessing step, the main algorithm runs in:

  $O(\Sigma_v (1 + \text{indeg}(v))) = O(n) + O(m) = O(n + m)$

  time.

- Therefore, total running time is $O(n + m)$

Further Improvements

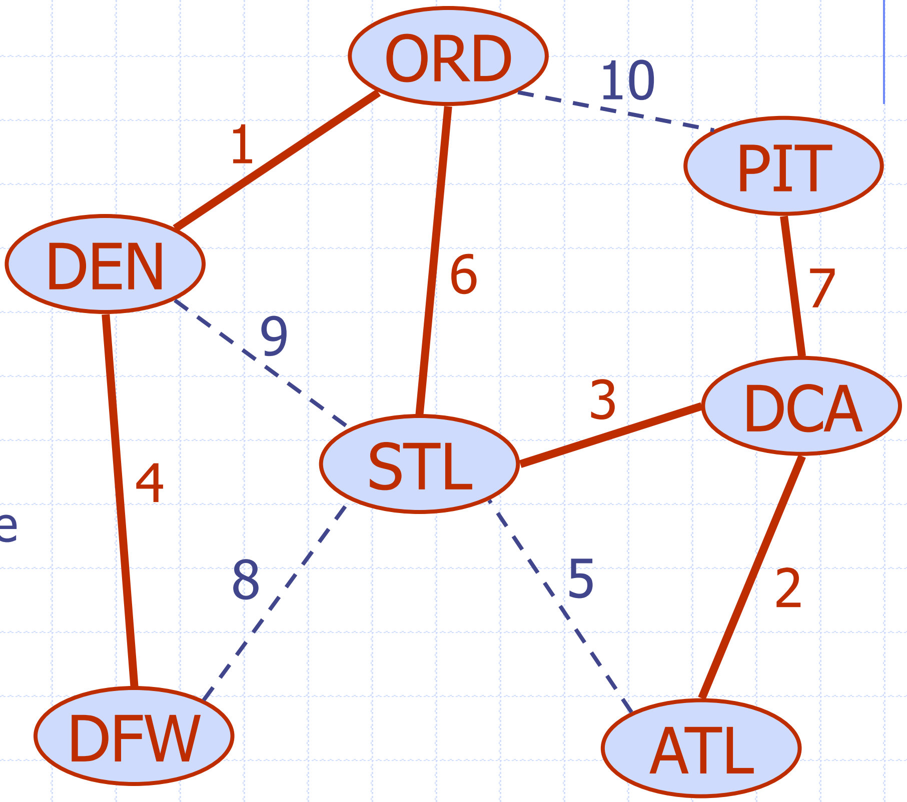
- Is there an efficient algorithm that computes the shortest path from a starting vertex s to a target t in a directed graph even if negative edge weights are allowed and even if the graph contains a directed cycle?
- Answer: Yes, as long as there are no *negative weight directed cycles*. The Bellman-Ford algorithm is similar to the dynamic programming version given here, but works with any directed graph, including those with a directed cycle. Bellman-Ford runs in O(|V|*|E|) in O(n^3) time. [Bellman-Ford is discussed in the book.]

Optional: Bellman-Ford

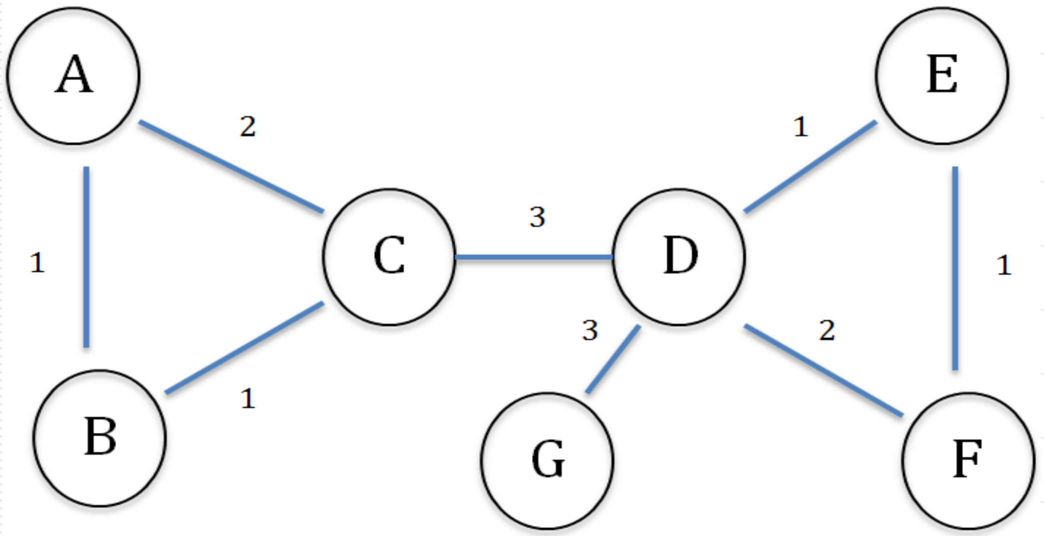
```
// Step 1: initialize graph
for each vertex v in vertices:
    if v is source then distance[v] := 0
    else distance[v] := inf
    predecessor[v] := null

// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w
            predecessor[v] := u

// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
return distance[], predecessor[]
```

Minimum Spanning Tree

- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Computer network (minimize cost of cable)
  - Transportation networks (minimize cost of road construction)



Kruskal's Greedy Strategy



Build a collection T of edges by doing the following: At each step, add an edge e to T of least weight subject to the constraint that adding e to T does not create a cycle in T. To answer questions about correctness and running time of this algorithm, we need to specify certain details.

Implementation Questions

1. How do we pick the next edge at each step?

2. How do we make sure that we do not add an edge to T that produces a cycle?

Solutions

1. We can arrange edges by sorting them by weight (in ascending order), and so we pick edges according to this sorted order.

2. We can ensure no cycles are created by building local minimum spanning trees around each vertex.

Kruskal's Algorithm

- First step is to sort all edges by weight.
- Second step involves creation of *clusters*
  - Every vertex is initially placed in a trivial *cluster* -- the cluster for a vertex v, denoted C(v), is simply {v}. A cluster represents a local minimum spanning tree.
  - When the next edge (u,v) is considered, C(u) and C(v) are compared -- if different, (u,v) is included as an edge in the final output tree, and C(u) and C(v) are merged.

Kruskal's Algorithm

Input: A simple connected weighted graph $G = (V, E)$ with n vertices and m edges
Output: A minimum spanning tree T of G

The Algorithm:

```
sort E in increasing order of edge weight
for each vertex $v$ in $G$, define an elementary cluster $C(v)$ (which will grow) by $C(v) = \{v\}$
$T \leftarrow$ an empty tree      // $T$ will eventually become the minimum spanning tree
while $T$ has fewer than $n - 1$ edges do
    $(u,v) \leftarrow$ next edge
    $C(v) \leftarrow$ cluster containing $v$
    $C(u) \leftarrow$ cluster containing $u$
    if $C(v) \neq C(u)$ then
        add edge $(u,v)$ to T
        merge $C(u)$ and $C(v)$  (and update other clusters as needed)
return $T$
```

Correctness

We need to verify the following Facts:

1. During execution, distinct clusters are always disjoint, and for each cluster C, if T[C] is the subgraph of G whose set of vertices is C and whose edges are those edges of T whose endpoints lie in C, then T[C] is a tree.
2. No cycle ever arises in T during execution of the algorithm
3. The main loop terminates (it is conceivable that after all edges have been examined, T still contains $< n - 1$ edges – this is shown to be impossible)
4. The set T that is returned is a spanning tree for G
5. The set T is a *minimum* spanning tree for G.

Verification of the Loop Invariant

To establish Facts 1 - 2, we establish the following loop invariant I(x,y) (for each edge (x,y) in E):

(a) Distinct clusters are disjoint

(b) For each cluster C, the subgraph T[C] is a tree

(c) T contains no cycles

[NOTE: As the algorithm proceeds, we do *not* expect T itself to be a tree because it is composed of disconnected pieces. We will show that these pieces do assemble into a tree by the time the algorithm has completed.]

At the beginning of execution, when singleton clusters are first formed, (a) – (c) hold. We now assume (a) – (c) hold and consider the iteration in which the next edge (x,y) is examined, and show that (a) – (c) continue to hold. In the algorithm, if C(x) = C(y), the iteration ends and the state of the clusters and that of the set T are unchanged, so we assume $C(x) \neq C(y)$.

To establish that (a) – (c) hold, we make use of the following results, established in Labs for the course:

Background Facts (from the Labs)

A. Suppose $V_1, V_2, \ldots, V_k$ are disjoint subsets of V and that $V = V_1 \cup V_2 \cup \ldots \cup V_k$. Then there is an edge (x,y) in E such that for some $i \neq j$, $x \in V_i$ and $y \in V_j$.

B. Suppose $S = (V_S, E_S)$ and $T = (V_T, E_T)$ are subtrees of G with no vertices in common (in other words, V_S and V_T are disjoint). Then for any edge (x,y) in E for which $x \in V_S$ and $y \in V_T$, the subgraph $U = (V_S \cup V_T, E_S \cup E_T \cup \{(x,y)\})$ is also a tree.

C. Suppose W is a subset of the set E of edges of G and $|W| < n - 1$. Consider the subgraph H of G formed from W by defining the edges of H to be W and defining the vertices of H to be the endpoints of those edges, and assume that H contains no cycle. Then there exists an edge (x,y) in G not in W so that the graph formed by $W \cup \{(x,y)\}$ also contains no cycle.

Verifying the Loop Invariant (a)-(c)

For (a) [distinct clusters are disjoint], joining two clusters with an edge, forming a new larger cluster, does not change the fact that distinct clusters are disjoint.

For (b) [T[C] is a tree], adding the edge (x,y) to the union of the trees T[C_x], T[C_y] results in another tree, T[C_x] U T[C_y] U {(x,y)}, by Background Fact B (previous slide). So, if we let C = C_x U C_y, then T[C] is a tree.

For (c) [no cycles], notice that the edges that compose T itself are formed as the union of the edges in the (disjoint) trees T[C_v], for v in V. Since none of these trees contains a cycle, T itself does not contain a cycle either.

This establishes the loop invariant and therefore Facts 1, 2.

Correctness – Fact 3, “while loop terminates”

We verify that the while loop in the algorithm eventually terminates:

Assume that after all edges have been examined, T still has < n-1 edges (this would be the situation when the while loop fails to terminate).

By Fact 2, T contains no cycle. By Background Fact 3, since |T| < n – 1 and contains no cycle, there is an edge e in G so that T U {e} also contains no cycles and so that e $\notin$ T.

But this situation is impossible: During the execution of Kruskal’s algorithm, the edge e = (x,y) was examined at some point, because, if the while loop does not terminate, *every* edge would be examined eventually. When e was examined, Kruskal rejected e (since e was not in T after all edges had been examined); the only reason Kruskal has for rejecting is to avoid creation of a cycle in T. But, by the way e was chosen above, clearly e does not introduce a cycle, as Background Fact 3 shows.

The reasoning shows that the assumption that the while loop never terminates is incorrect.

Correctness – Fact 4, "T is a spanning tree"

We verify Fact 4, that, after execution of Kruskal's algorithm, T is a spanning tree.

- We have already seen that T has no cycles and therefore (by the way the while loop is defined) T has $n - 1$ edges, where n is the number of vertices in G. By the lemma below, T must have n vertices. It follows that T is a spanning tree.

Lemma. If a graph H is a graph that has n vertices and m edges and if $m \geq n$, then H has a cycle.

Proof. If H is connected but has no cycle, it follows that $m = n - 1$. Since $m \geq n$, it therefore follows that H has a cycle.

If H is not connected, write H as a union of its connected components: $H = H_1 \cup H_2 \cup \ldots \cup H_k$, where $k > 1$, where, for each i, H_i has n_i vertices and m_i edges. Since each H_i is a tree, $m_i = n_i - 1$. We have therefore

$$m = m_1 + \ldots + m_k = (n_1-1) + \ldots + (n_k-1) = n - k < n,$$

contradiction.

Optional: Correctness – Fact 5

To verify Fact 5 – that, after execution of Kruskal's algorithm, T is a *minimum* spanning tree – the main idea is to establish the following two points

- At each stage of the algorithm, for each cluster C, T[C] is not only a tree, but is in fact a minimum spanning tree for G[C], the subgraph induced by C.
- After the algorithm has finished, all clusters have merged into a single cluster, which must be equal to V, the set of all vertices of G.

From these two points, it follows that T = T[V] is a minimum spanning tree for G = G[V].

To establish the truth of these two points, we consider the following enhanced loop invariant:

<u>I(e)</u>: *For each cluster C, T[C] is a minimum spanning tree for G[C].*

Correctness (Fact 5)

Before the first iteration, I is clearly true. After the first iteration, when the first edge is added and two clusters merge, it is still true. We consider later iterations inductively: We assume I(i) and prove I(i+1). We assume that at the i+1st iteration, an edge e = (x,y) is added; if not, I(i+1) holds trivially.

Let C, D be the clusters for which $x \in C$ and $y \in D$. We show that $T_{C,D}$ = T[C] U T[D] U {e} is an MST for $G[V_1,V_2]$. If not, let S be an MST for $G[V_1,V_2]$.

Then S[C] (S restricted to the vertices of C) is a spanning tree for G[C], and S[D] is a spanning tree for G[D]. Total weight of S[C] is $\geq$ total weight of T[C] and total weight of S[D] is $\geq$ total weight of T[D] (by minimality of the spanning trees T[C], T[D]). Since both $T_{C,D}$ and S are spanning trees for $G[V_1,V_2]$, they have the same number of edges. Likewise, T[C] and S[C] must have the same number of edges, and T[D] and S[D] must have the same number of edges. This means that there is just one edge f in S having one endpoint in C and the other in D. Since total weight of S is less than that of $T_{C,D}$ (since S is truly an MST) we have

$$wt(f) < wt(e).$$

Correctness, Fact 5 (continued)

It follows that f was processed by the algorithm before e was processed, and was rejected because both its endpoints already belong to a single cluster; but this is impossible because one endpoint of f lies in C, the other in D, and C, D, at this stage, are disjoint clusters. Contradiction. We have shown I(i+1) holds, assuming that I(i) does.

To complete the proof of Fact 5, we need to show that all the local MSTs eventually merge into a single MST for G. Since, as we have shown, at any stage of the algorithm, T is the union of the trees T[C], for all clusters C, if at the end of the algorithm there were two (or more) clusters C_1, C_2 remaining, then T would be the union of two (or more) disjoint trees, and would therefore be disconnected. But T has been shown to be a tree (and is therefore connected). This shows all the clusters have merged into one cluster by the end of the algorithm.

Running Time of Kruskal: First Try

- Time to sort edges: O(mlog n)
- Cost of while loop = O(mn)
  - loop potentially accesses every edge
  - comparison C(x) = C(y), with a hashtable implementation of sets, is O(n)
  - merging C(x), C(y) costs min{|C(x)|, |C(y)|}, which is O(n).
- Cost of while loop can be improved with a good choice of data structure

DisjointSets Data Structure

- Data structure for maintaining a partition of a set into disjoint subsets (data structure sometimes called *Partition* rather than DisjointSets)
- General features
  - *Data:*
    - Universe U – the base set that is being partitioned (this set is never altered)
    - Collection C = $\{X_1, X_2, \ldots, X_n\}$ of subsets of the universe – the subsets are disjoint and their union is U (these subsets are modified when the data structure is used – size of C shrinks because of repeated union operations)
  - *Operations:*
    - find(x) – returns the subset X_i to which x belongs
    - union(A,B) – replaces the subsets A, B in C with A U B.

Tree-Based Implementation of DisjointSets

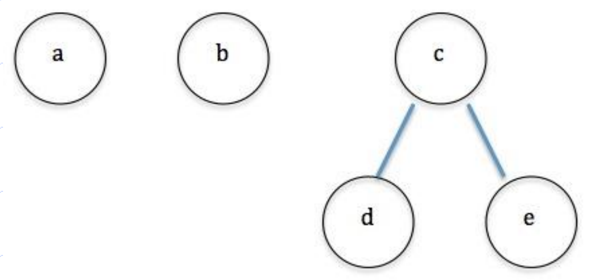
- The elements of each set X in the collection C are represented by nodes in a tree T_X; the set X itself is referenced by its root r_X.
- find(x) returns the root of the tree to which x belongs
- union(x,y) joins the tree x belongs to to the tree y belongs to by pointing root of one to the root of the other.

Example

U = {'a', 'b', 'c', 'd', 'e'}

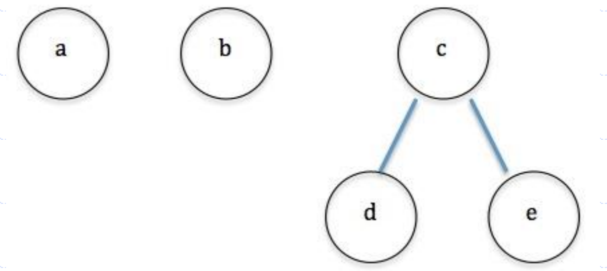
C = {{'a'}, {'b'}, {'c', 'd', 'e'}}

Tree representations:

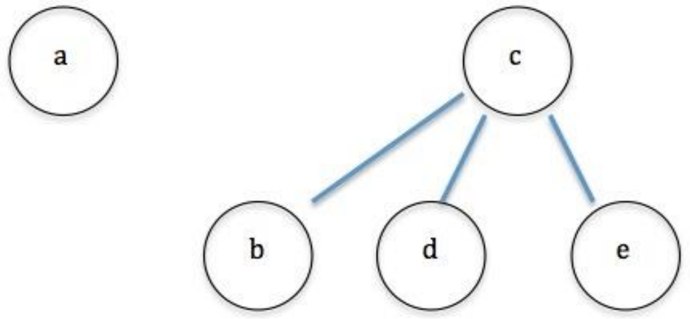


- find('d') returns 'c'

Example (cont)



- union('b', 'c') points root 'b' to root 'c'



Now, find('b') returns 'c'

Code

```
//handle trees by keeping track of parents only
//whenever a character is a root, its parent is set to be itself
HashMap<Character, Character> parents = new HashMap<Character, Character>();
char[] universe;

//find returns the root of tree representing a subset
//worst case: find requires full depth of tree to locate root of representing tree
public char find(char element) {
    char nextParent = parents.get(element);
    if(nextParent == element) {
        return element;
    } else {
        return find(nextParent);
    }
}
```

Code

```
//union() accepts only tree roots (representing subsets) as arguments
//The method simply points the first root to the second
//In the worst case, resulting tree is taller than original two
public void union(char a_tree, char b_tree) {
    parents.put(a_tree, b_tree);
}
```

To avoid building up trees that are too tall (and therefore imbalanced), an optimization can be used: Always point the shorter tree's root to that of the taller.

Optimized Code

```
HashMap<Character, Character> parents = new HashMap<Character, Character>();
char[] universe;
//keep track of heights of trees
HashMap<Character, Integer> heights = new HashMap<Character, Integer>();

public void union(char a_tree, char b_tree) {
    int height_a = heights.get(a_tree);
    int height_b = heights.get(b_tree);
    if(height_a < height_b) {
        parents.put(a_tree, b_tree);
    }  else if(height_b < height_a) {
        parents.put(b_tree, a_tree);
    }  else { //height_a == height_b
        parents.put(a_tree, b_tree);
        heights.put(b_tree, height_b + 1); //this is case in which height is increased
    }
}
```

NOTE: With this optimization of union(), find() can be shown to run in O(log n) in the worst case.

Optimized Running Time of Kruskal

- Time to sort edges: O(mlog n)
- Cost of while loop = O(mlog n)
  - loop potentially accesses every edge //O(m)
  - comparison C(x) = C(y) follows these steps:
    //locates roots of representing trees
    - $r_x \leftarrow$ find(x) and $r_y \leftarrow$ find(y) //O(log n)
    - check whether $r_x = r_y$ //O(1)
  - merging C(x), C(y) is done by union() operation //O(1)

- Total (optimized) running time for Kruskal: O(mlog n).

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. A Minimum Spanning Tree can be obtained from a weighted graph G = (V,E) by examining all possible subgraphs of G, and extracting from those that are trees having the smallest sum of edge weights. This procedure runs in $\Omega(2^m)$, where n = |E|.
2. Kruskal's Algorithm is a highly efficient procedure (O(mlog n)) for finding an MST in a graph G. It proceeds by choosing edges with minimum possible weight subject to the constraint that selected edges do not introduce a cycle in the set T of edges obtained so far.

3. *Transcendental Consciousness*, the simplest form of awareness, is the source of effortless right action.
4. *Impulses Within the Transcendental Field.* Effortless, economical, mistake-free creation arises from the self-referral dynamics of the field of pure consciousness.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, optimal solutions arise as an effortless unfoldment within one's unbounded nature.