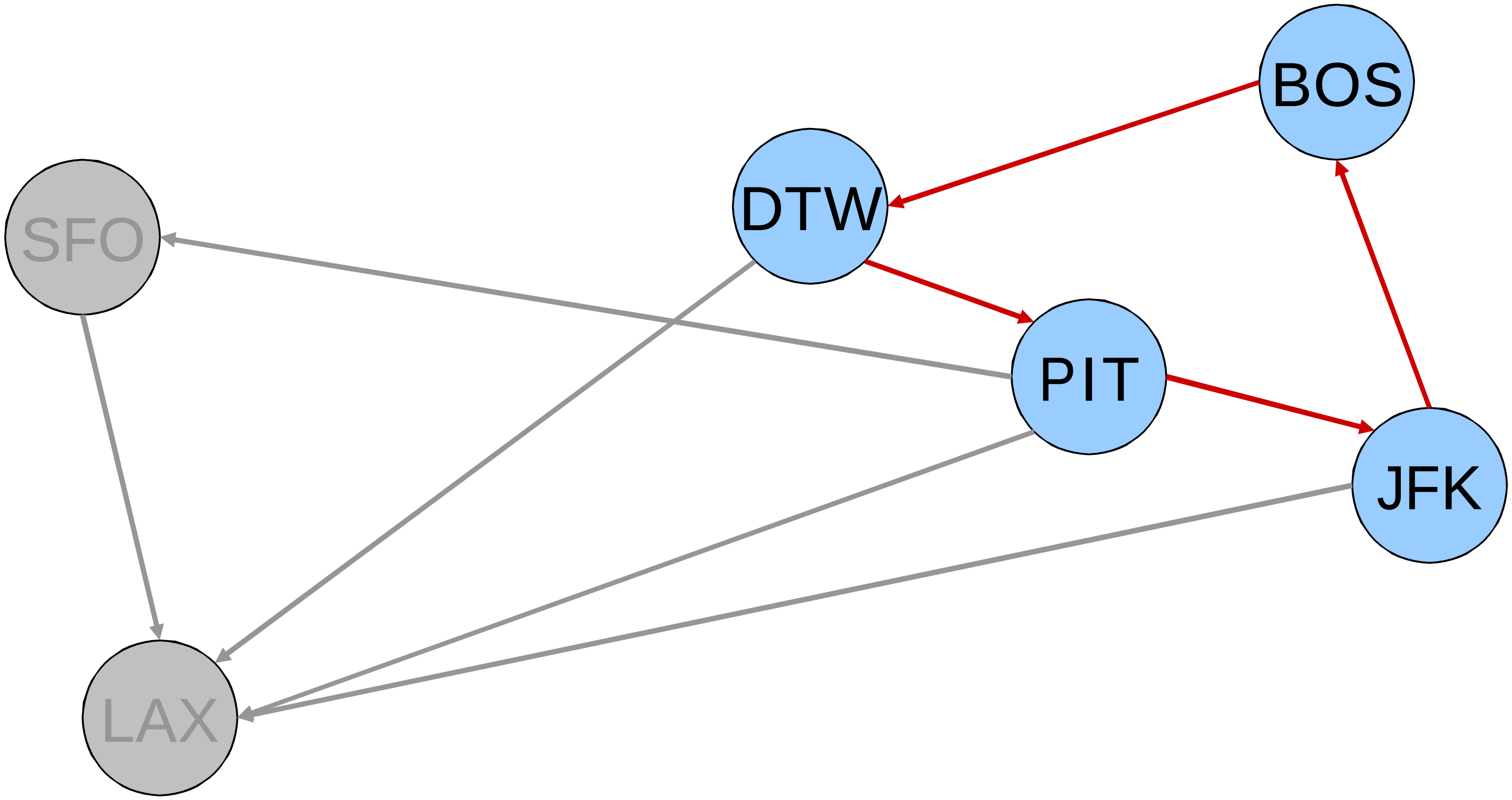
DATA STRUCTURES USING ‘C’

Paths and cycles

- A *path* is a sequence of nodes v_1, v_2, …, v_N such that $(v_i,v_{i+1}) \in E$ for $0<i<N$
  - The *length* of the path is N-1.
  - *Simple path*: all v_i are distinct, $0<i<N$

- A *cycle* is a path such that $v_1=v_N$
  - An *acyclic* graph has no cycles

Cycles



More useful definitions

- In a directed graph:

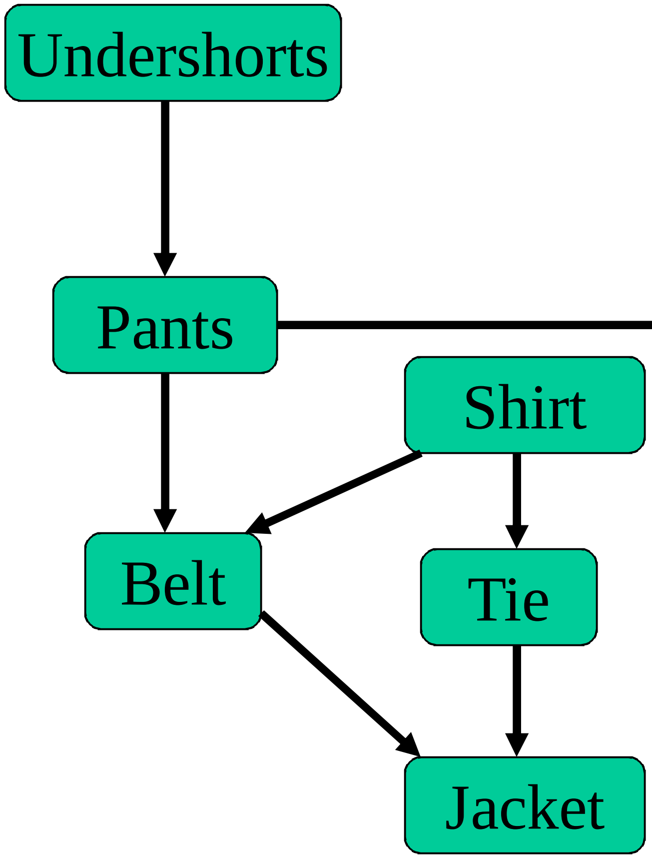
- The *indegree* of a node v is the number of distinct edges (w,v)∈E.
- The *outdegree* of a node v is the number of distinct edges (v,w)∈E.

- A node with indegree 0 is a *root*.

Trees are graphs

- A *dag* is a directed acyclic graph.
- A *tree* is a connected acyclic undirected graph.
- A *forest* is an acyclic undirected graph (not necessarily connected), i.e., each connected component is a tree.

Example DAG


Undershorts

Socks

Watch

Pants

Shoes

Shirt

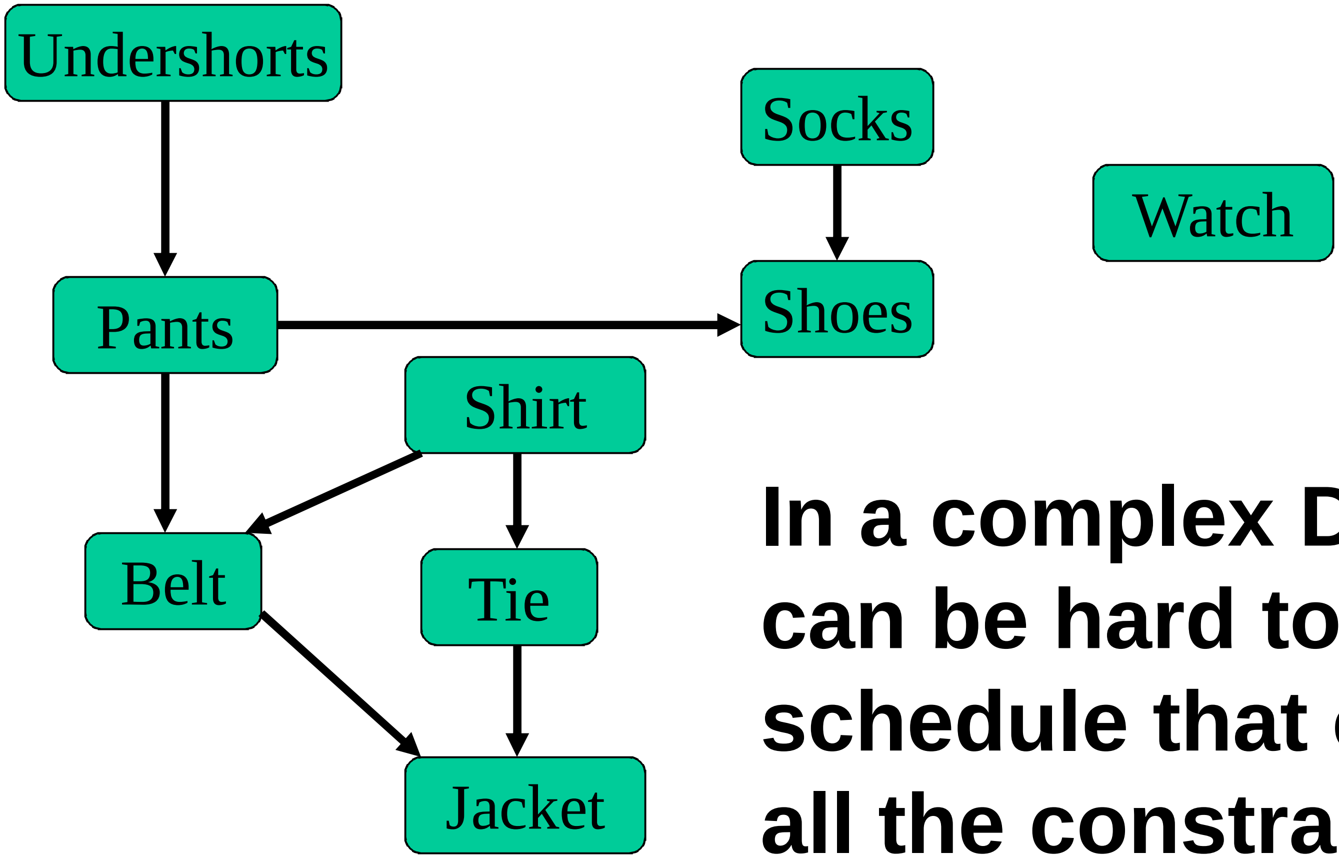
Belt

Tie

Jacket

a DAG implies an ordering on events

Example DAG



In a complex DAG, it can be hard to find a schedule that obeys all the constraints.

Topological Sort

Topological Sort

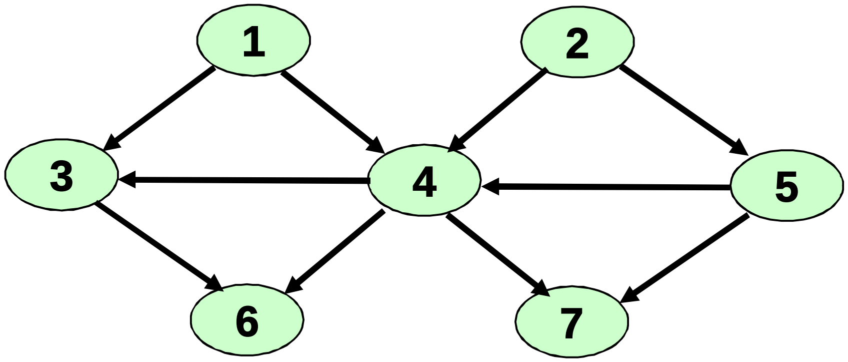
- For a directed acyclic graph G = (V,E)
- A topological sort is an ordering of all of G's vertices $v_1, v_2, \ldots, v_n$ such that...

Formally: for every edge (v_i,v_k) in *E*, $i<k$.

Visually: all arrows are pointing to the right

Topological sort

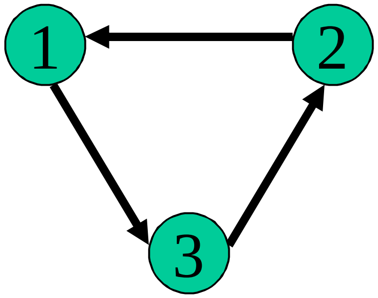
- There are often many possible topological sorts of a given DAG
- Topological orders for this DAG :
    - 1,2,5,4,3,6,7
    - 2,1,5,4,7,3,6
    - 2,5,1,4,7,3,6
    - *Etc.*



- Each topological order is a *feasible schedule*.

Topological Sorts for Cyclic Graphs?

Impossible!



- If v and w are two vertices on a cycle, there exist paths from v to w *and* from w to v.
- Any ordering will contradict one of these paths

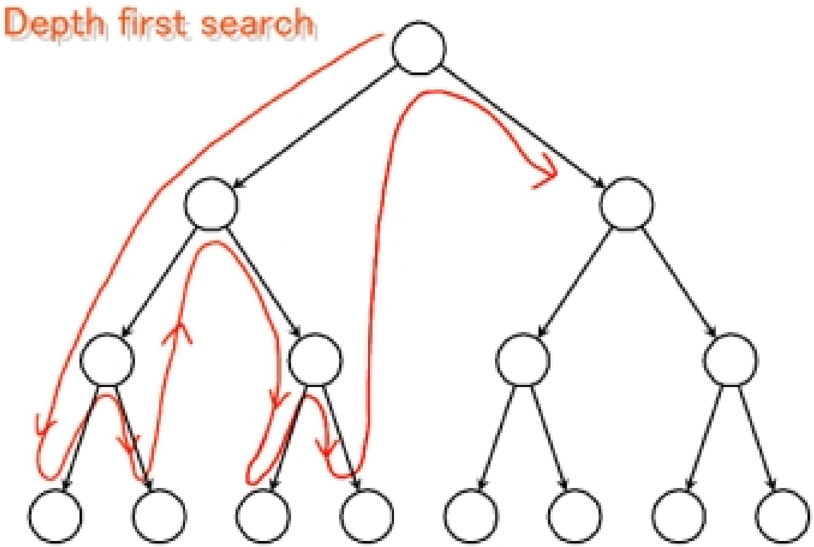
Topological sort algorithm

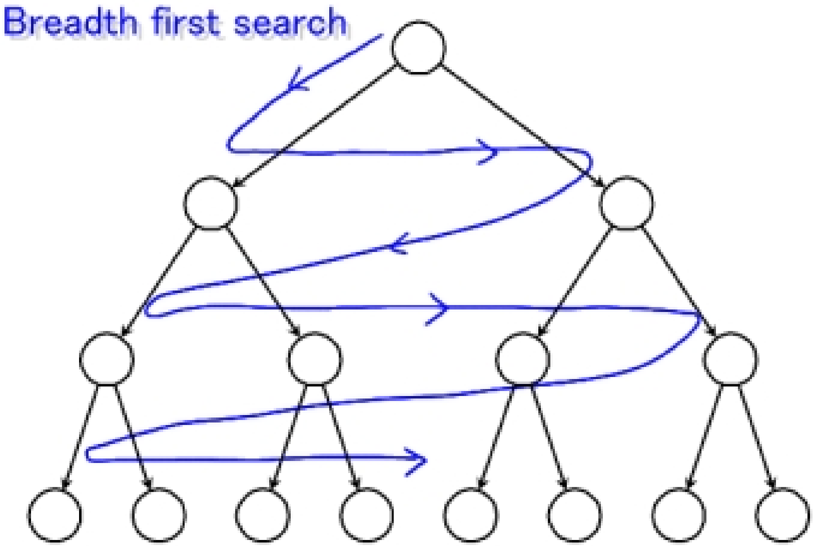
- Algorithm
  - Assume indegree is stored with each node.
  - Repeat until no nodes remain:
    - Choose a root and output it.
    - Remove the root and all its edges.

- Performance
  - $O(V^2 + E)$, if linear search is used to find a root.

Graph Traversals

Graph Traversals





- Both take time: O(V+E)

Use of a stack

- It is very common to use a stack to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a stack leads to a *depth-first* visit order.
- Depth-first visit order is “aggressive” in the sense that it examines complete paths.

Topological Sort as DFS

- do a DFS of graph G
- as each vertex v is “finished” (all of it’s children processed), insert it onto the front of a linked list
- return the linked list of vertices

- *why is this correct?*

Use of a queue

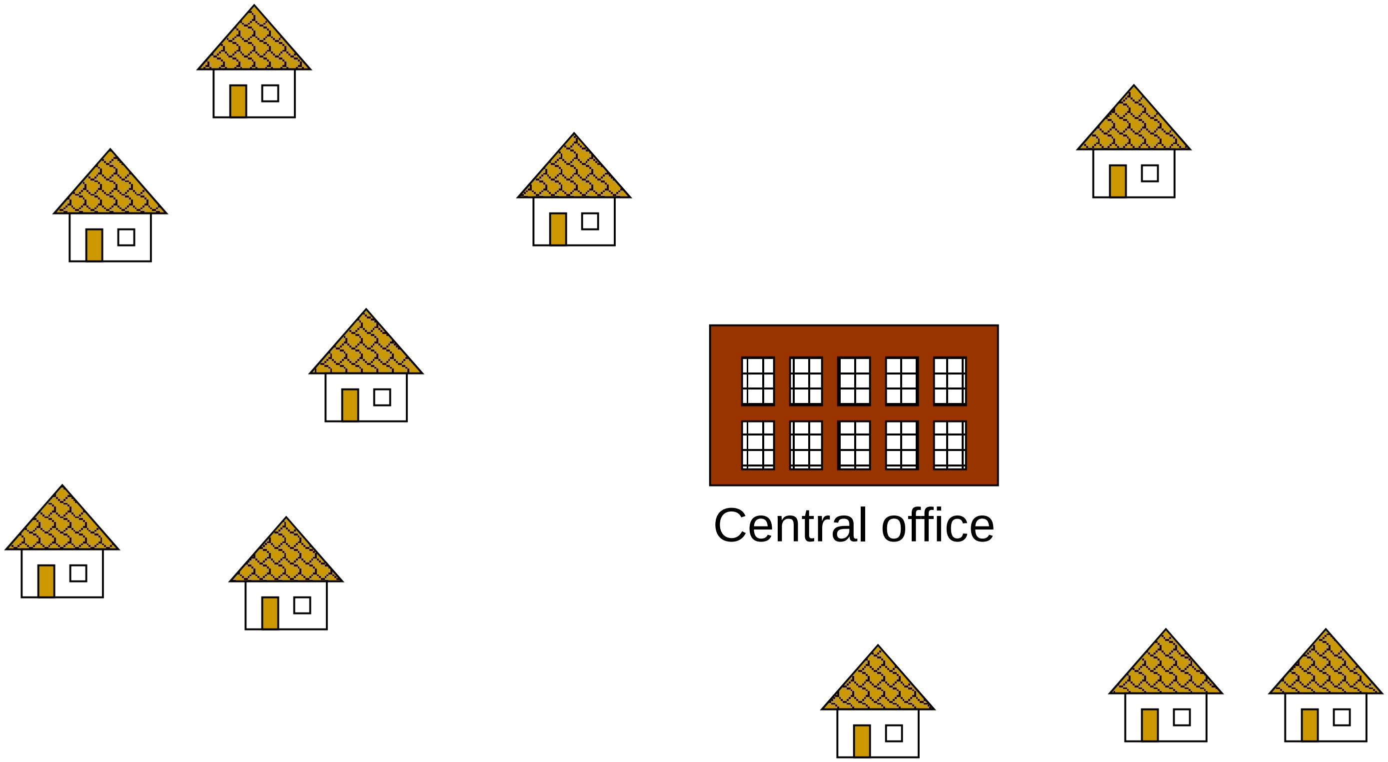
- It is very common to use a queue to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a queue leads to a *breadth-first* visit order.
- Breadth-first visit order is “cautious” in the sense that it examines every path of length i before going on to paths of length i+1.

Graph *Searching ???*

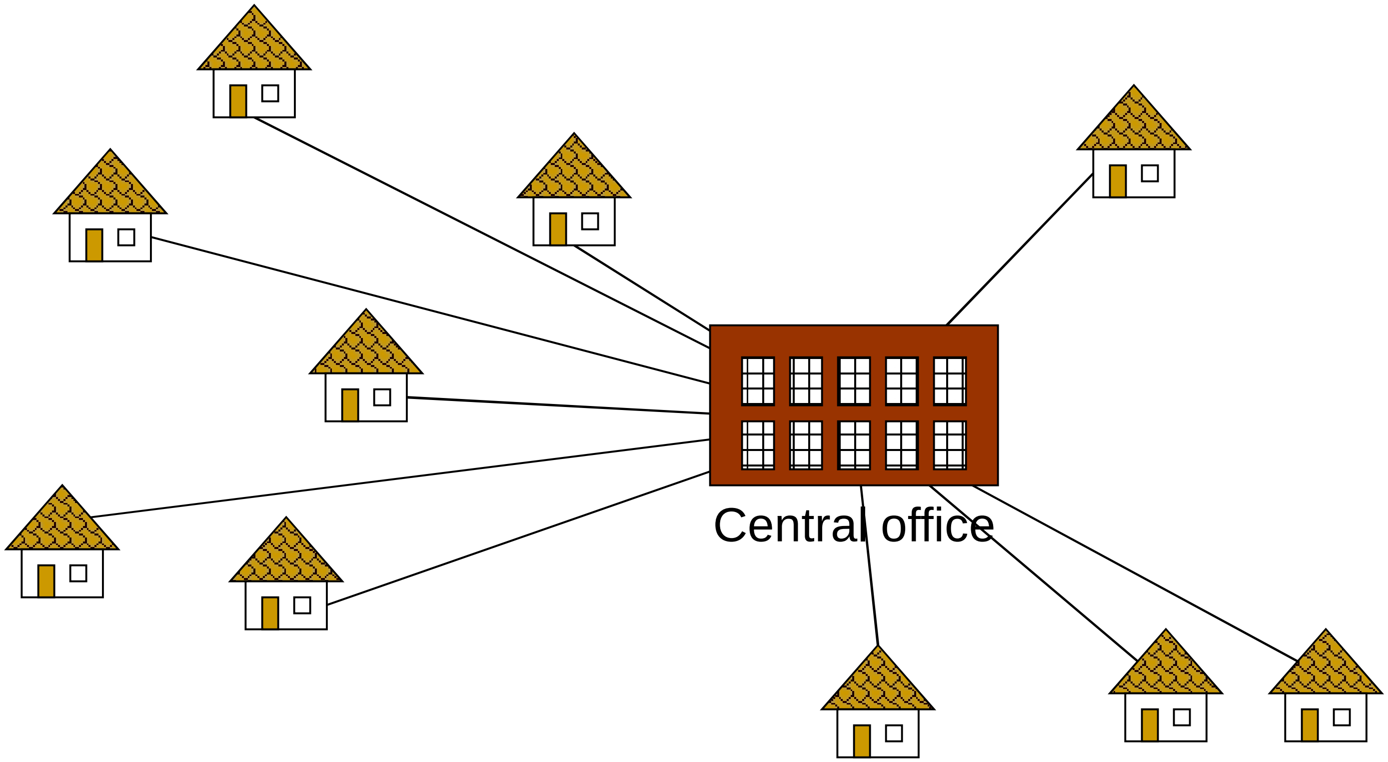
- Graph as state space (node = state, edge = action)
- For example, game trees, mazes, ...
- BFS and DFS each search the state space for a best move. If the search is exhaustive they will find the same solution, but if there is a time limit and the search space is large...
- DFS explores a few possible moves, looking at the effects far in the future
- BFS explores many solutions but only sees effects in the near future (often finds shorter solutions)

Minimum Spanning Trees

Problem: Laying Telephone Wire

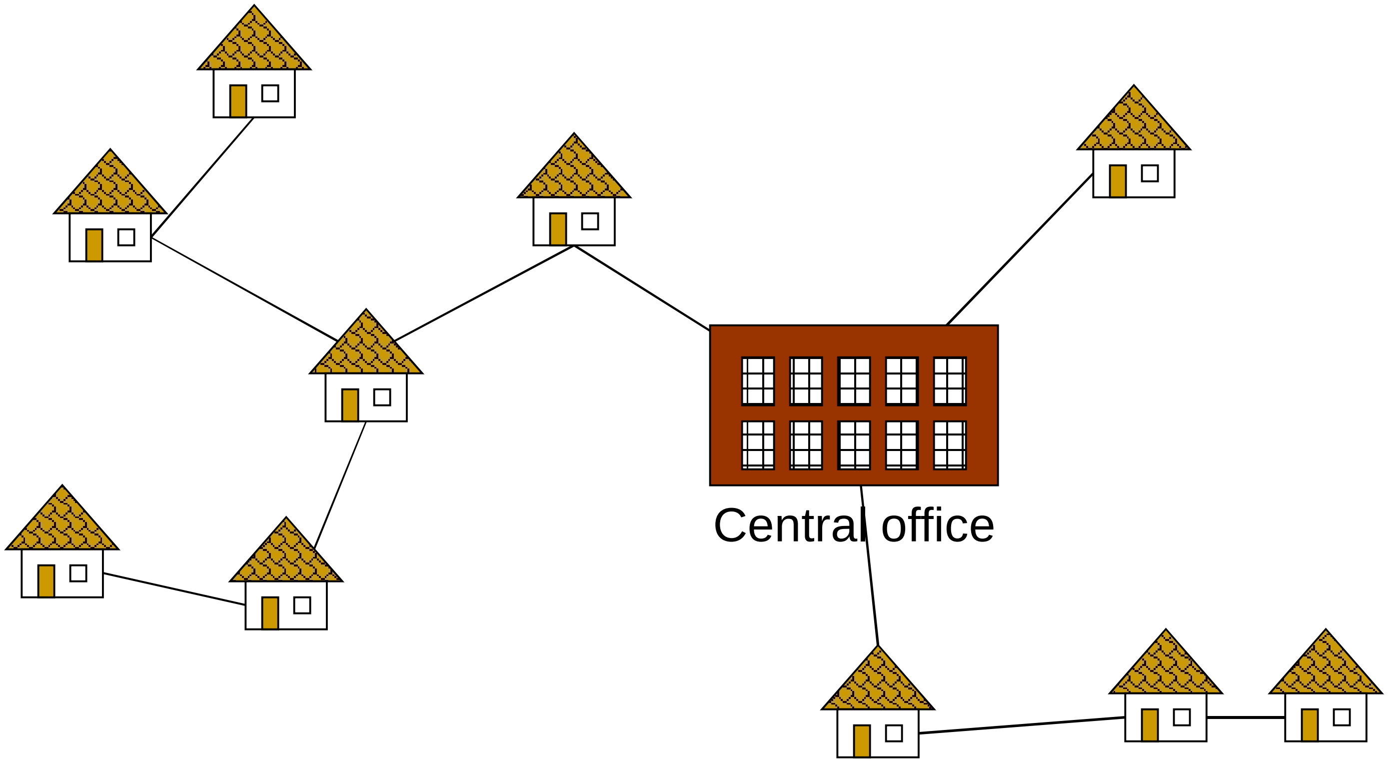


Wiring: Naïve Approach



Expensive!

Wiring: Better Approach



Minimize the total length of wire connecting the customers

Minimum Spanning Tree (MST)

(see Weiss, Section 24.2.2)

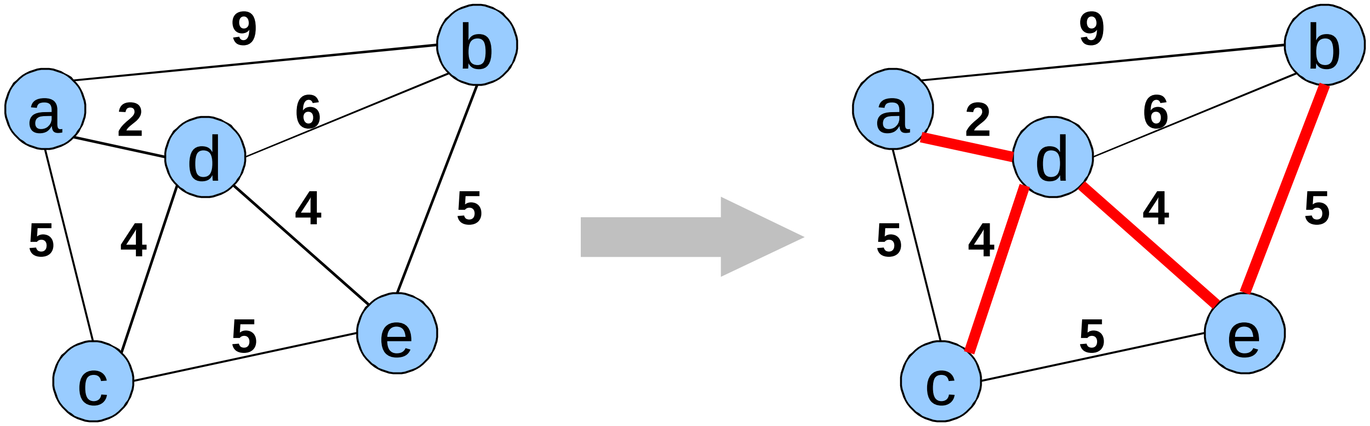
A **minimum spanning tree** is a subgraph of an undirected weighted graph G, such that

- it is a tree (i.e., it is acyclic)
- it covers all the vertices V
  - contains $|V| - 1$ edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique

Applications of MST

- Any time you want to visit all vertices in a graph at minimum cost (e.g., wire routing on printed circuit boards, sewer pipe layout, road planning…)
- Internet content distribution
  - $$$, also a hot research topic
  - Idea: publisher produces web pages, content distribution network replicates web pages to many locations so consumers can access at higher speed
  - MST may not be good enough!
    - content distribution on minimum cost tree may take a long time!
- Provides a heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree (*why??*)

How Can We Generate a MST?



Prim’s Algorithm

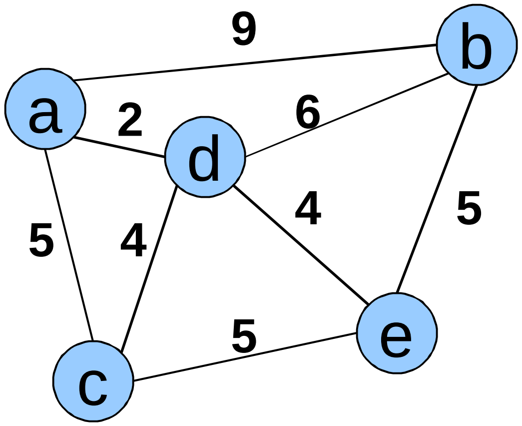
Initialization

a. Pick a vertex r to be the root

b. Set $D(r) = 0$, $parent(r) = null$

c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$

d. Insert all vertices into priority queue P, using distances as the keys



e	a	b	c	d
0	∞	∞	∞	∞

Vertex	Parent
e	-

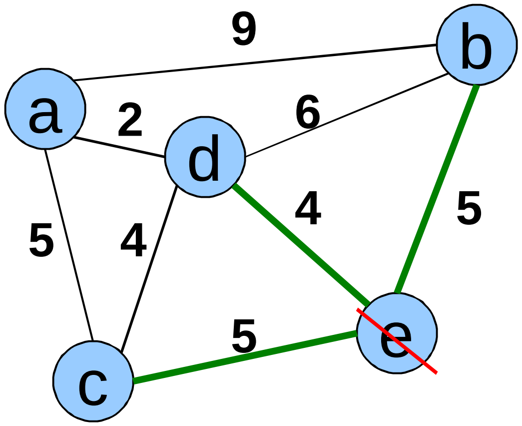
Prim’s Algorithm

```
While $P$ is not empty:

   1. Select the next vertex $u$ to add to the tree
         $u = P.deleteMin()$

   2. Update the weight of each vertex $w$ adjacent to
      $u$ which is not in the tree (i.e., $w \in P$)
         If $weight(u,w) < D(w)$,
            a. $parent(w) = u$
            b. $D(w) = weight(u,w)$
            c. Update the priority queue to reflect
                 new distance for $w$
```

Prim’s algorithm



e	d	b	c	a
0	∞	∞	∞	∞

Vertex	Parent
e	-
b	-
c	-
d	-

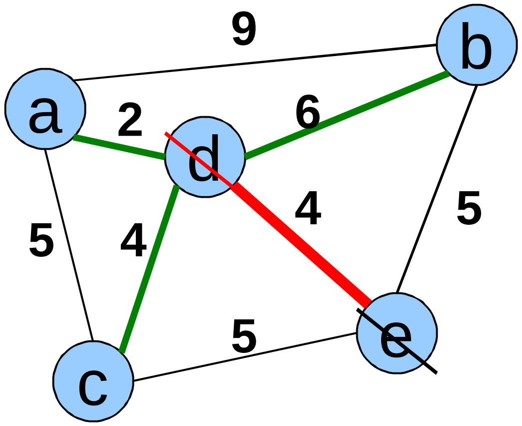
---

d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex ***e,*** and we update the distances and parent for its adjacent vertices

Prim’s algorithm



d	b	c	a
4	**5**	**5**	∞

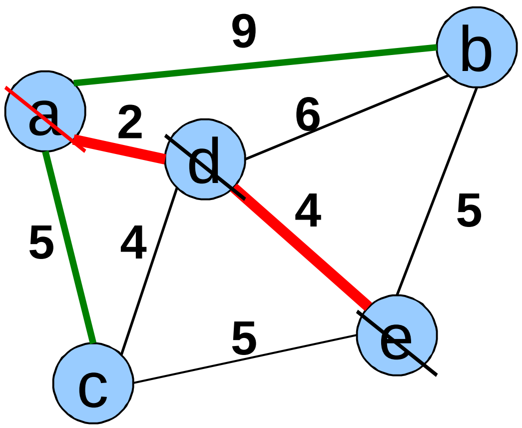
Vertex	Parent
e	-
b	e
c	e
d	e

---

a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim’s algorithm



a	c	b
2	**4**	**5**

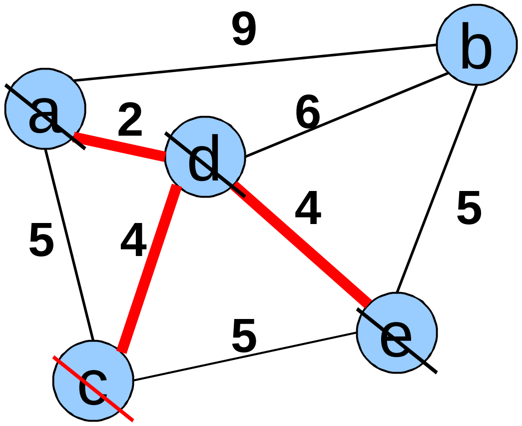
Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

c	b
4	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim’s algorithm



c	b
4	**5**

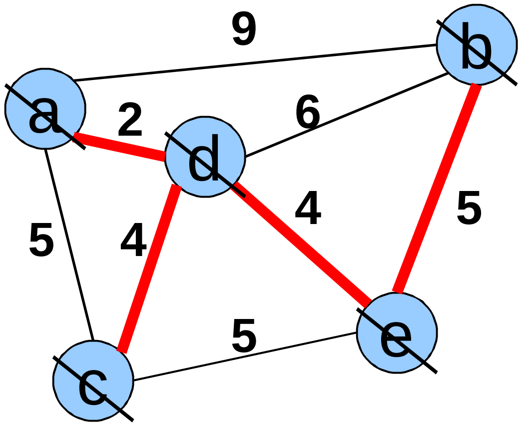
Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim’s algorithm



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Running time of Prim’s algorithm (without heaps)

Initialization of priority queue (array): O(|V|)

Update loop: |V| calls

- Choosing vertex with minimum cost edge: O(|V|)
- Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of O(|E|) updates

Overall cost without heaps: $\boxed{O(|E| + |V|^2)}$

When heaps are used, apply same analysis as for Dijkstra’s algorithm (p.469) (*good exercise*)

Prim's Algorithm Invariant

- At each step, we add the edge (***u,v***) s.t. the weight of (***u,v***) is **minimum** among all edges where ***u*** is in the tree and ***v*** is not in the tree

- Each step maintains a minimum spanning tree of the vertices that have been included thus far

- When all vertices have been included, we have a MST for the graph!

Correctness of Prim's

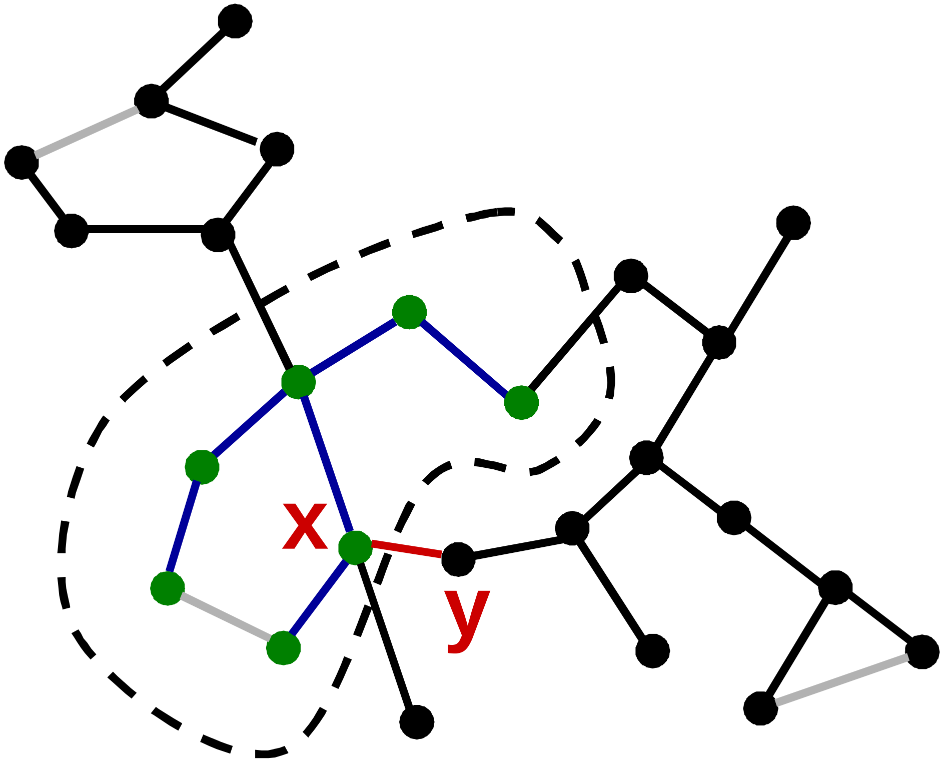
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

But is this a *minimum* spanning tree?

Suppose it wasn't.

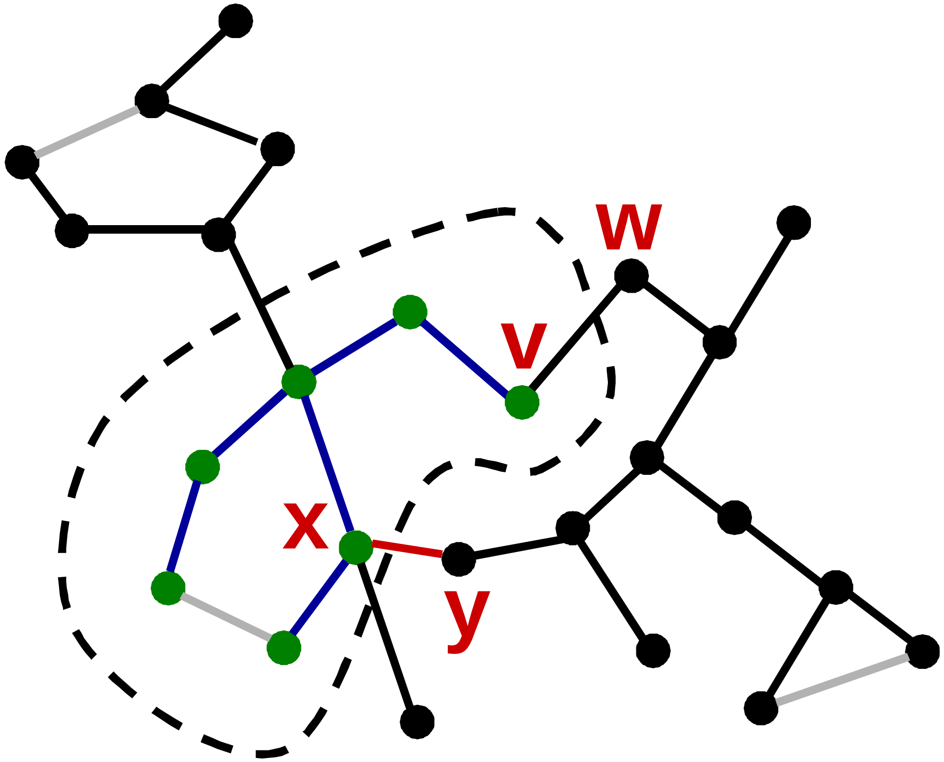
- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

Correctness of Prim’s



- Let *G* be a connected, undirected graph
- Let **S** be the set of edges chosen by Prim’s algorithm *before* choosing an errorful edge **(*x*,*y*)**
- Let **V'** be the vertices incident with edges in **S**
- Let **T** be a MST of *G* containing all edges in **S**, but not **(*x*,*y*)**.

Correctness of Prim's



- Edge **(*x*,*y*)** is not in **T**, so there must be a path in **T** from ***x*** to ***y*** since **T** is connected.
- Inserting edge **(*x*,*y*)** into **T** will create a cycle
- There is exactly one edge on this cycle with exactly one vertex in **V'**, call this edge **(*v*,*w*)**

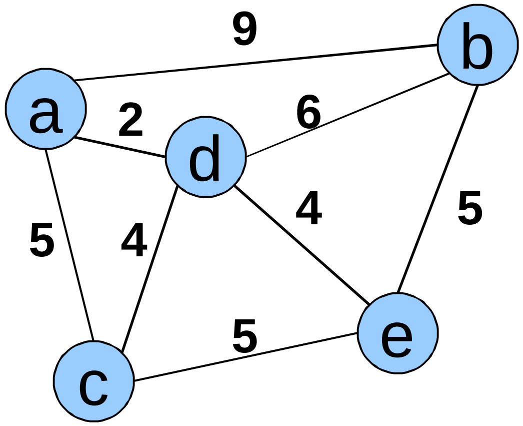
Correctness of Prim’s

- Since Prim’s chose **(*x*,*y*)** over **(*v*,*w*)**, w(**v**,**w**) >= w(**x**,**y**).
- We could form a new spanning tree **T’** by swapping **(*x*,*y*)** for **(*v*,*w*)** in **T** (*prove this is a spanning tree*).
- w(**T’**) is clearly no greater than w(**T**)
- But that means **T’** is a MST
- And yet it contains all the edges in **S**, and also **(*x*,*y*)**

...Contradiction

Another Approach

- Create a forest of trees from the vertices
- Repeatedly merge trees by adding "**safe edges**" until only one tree remains
- A "safe edge" is an edge of minimum weight which does not create a cycle



forest: {a}, {b}, {c}, {d}, {e}

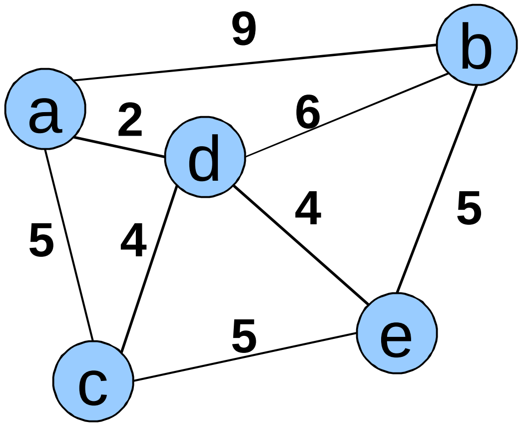
Kruskal’s algorithm

Initialization

a. Create a set for each vertex $v \in V$

b. Initialize the set of “safe edges” A comprising the MST to the empty set

c. Sort edges by increasing weight



F = {a}, {b}, {c}, {d}, {e}

$A = \varnothing$

E = {(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)}

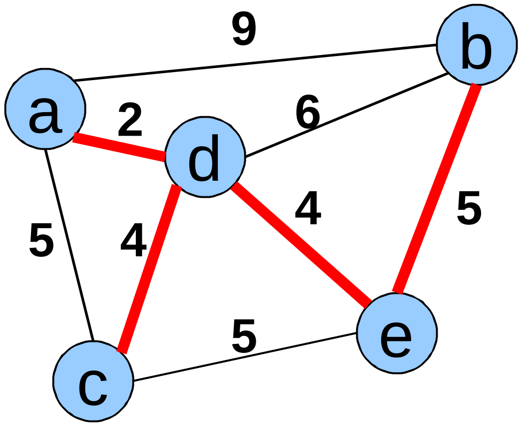
Kruskal’s algorithm

```
For each edge $(u,v) \in E$ in increasing order
while more than one set remains:
    If $u$ and $v$, belong to different sets $U$ and $V$
        a. add edge $(u,v)$ to the safe edge set
            $A = A \cup \{(u,v)\}$
        b. merge the sets $U$ and $V$
            $F = F - U - V + (U \cup V)$

Return $A$
```

- Running time bounded by sorting (or findMin)
- $O(|E|\log|E|)$, or equivalently, $O(|E|\log|V|)$ (***why???***)

Kruskal’s algorithm



E = {(~~a,d~~), (~~c,d~~), (~~d,e~~), (~~a,c~~),
(~~b,e~~), (c,e), (b,d), (a,b)}

Forest	*A*
{a}, {b}, {c}, {d}, {e}	∅
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

Kruskal’s Algorithm Invariant

- After each iteration, every tree in the forest is a MST of the vertices it connects
- Algorithm terminates when all vertices are connected into one tree

Correctness of Kruskal's

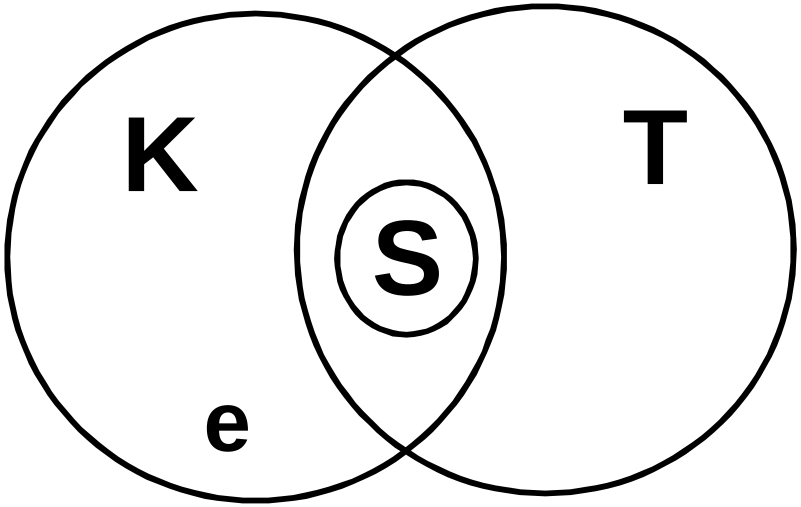
- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

But is this a *minimum* spanning tree?

Suppose it wasn't.

- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.

Correctness of Kruskal’s



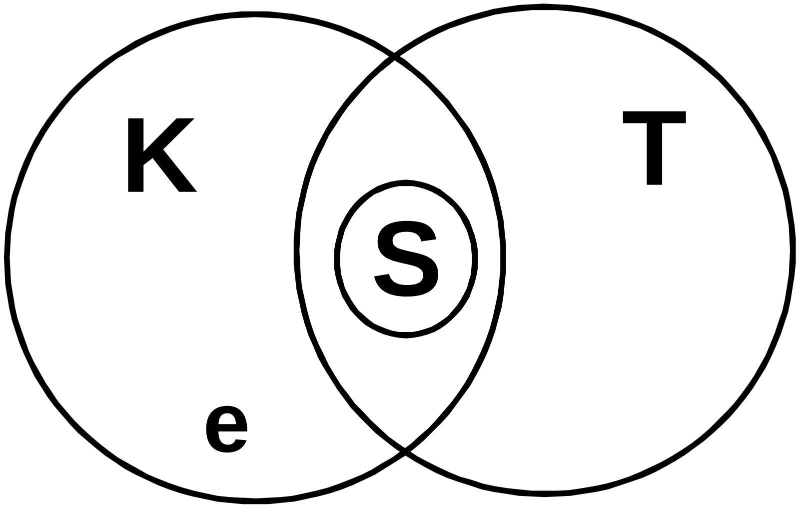
- Let **e** be this first errorful edge.
- Let **K** be the Kruskal spanning tree
- Let **S** be the set of edges chosen by Kruskal’s algorithm *before* choosing **e**
- Let **T** be a MST containing all edges in **S**, but not **e**.

Correctness of Kruskal's

Lemma: w(**e'**) >= w(**e**) for all edges **e'** in **T - S**

Proof (*by contradiction*):

- Assume there exists some edge **e'** in **T - S**, w(**e'**) < w(**e**)
- Kruskal's must have considered **e'** before **e**



- However, since **e'** is not in **K** (*why??*), it must have been discarded because it caused a cycle with some of the other edges in **S**.
- But **e' + S** is a subgraph of **T**, which means it cannot form a cycle **...Contradiction**

Correctness of Kruskal’s

- Inserting edge **e** into **T** will create a cycle
- There must be an edge on this cycle which is not in **K** (*why??*). Call this edge **e’**
- **e’** must be in **T - S,** so (by our lemma) w(**e’**) >= w(**e**)
- We could form a new spanning tree **T’** by swapping **e** for **e’** in **T** (*prove this is a spanning tree*).
- w(**T’**) is clearly no greater than w(**T)**
- But that means **T’** is a MST
- And yet it contains all the edges in **S**, and also **e**

...Contradiction

Greedy Approach

- Like Dijkstra’s algorithm, both Prim’s and Kruskal’s algorithms are **greedy algorithms**
- The greedy approach works for the MST problem; however, **it does not work for many other problems**!

That’s All!