# Graph Theory

A la CS61b we discuss a few graph theory things...

## Notation

A graph $G = (V, E)$ is an ordered pair.

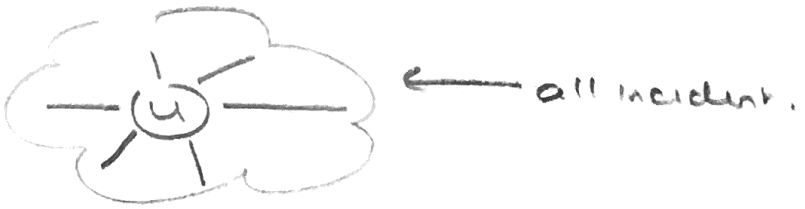
$V \equiv$ { set of nodes }

$E \equiv$ { set of edges }.

Edges can be ordered $(u, v)$ represents...

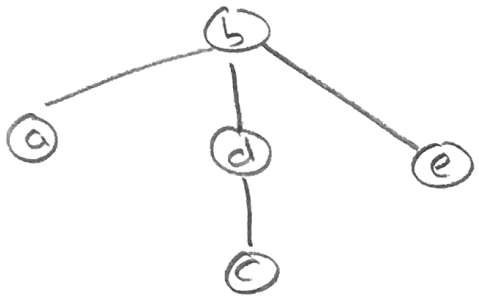

implying $G$ is directed (a digraph) or unordered $(u, v)$.


we say edges are incident to their nodes if they touch it...




If a graph has a cycle, it is cyclic, else it is acyclic.
(only true if it is directed) (directed acyclic graph).

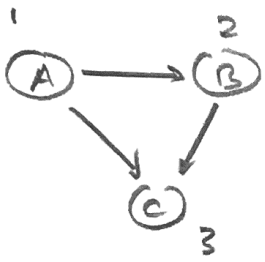
trees are acyclic graphs... (assumed directed).

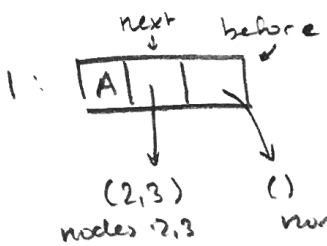


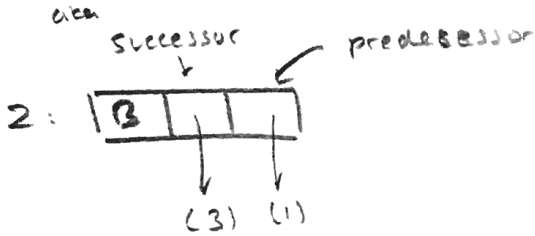

these are also connected ie there is a path between every pair of nodes.

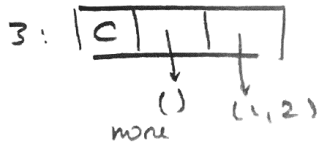
## Representation

Edge lists...



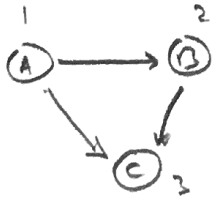




for each node you store what it leads to by an edge and what nodes lead to it.

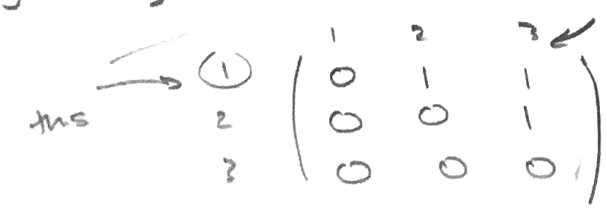
Edge sets:

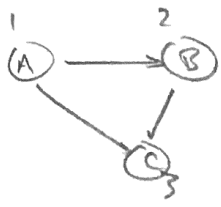
just a random collection of edges.




$\{(1,2),(2,3),(1,3)\}$

Adjacency matrix ⟶ connects to



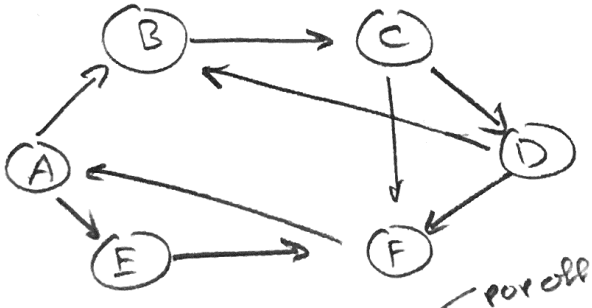
# Traversing a Graph

Here we use one generic algorithm to implement three algorithms.

```
search (G, u):
    fringe = <insert list object>.
    add u to fringe.
    while fringe is not empty:
        v = first element of fringe.
        if v not visited:
            visit v
            add all neighbors of v to fringe
    return traversal.
```

Depending on what list object you use, you get one of three algorithms...

① Stack → Depth First search.



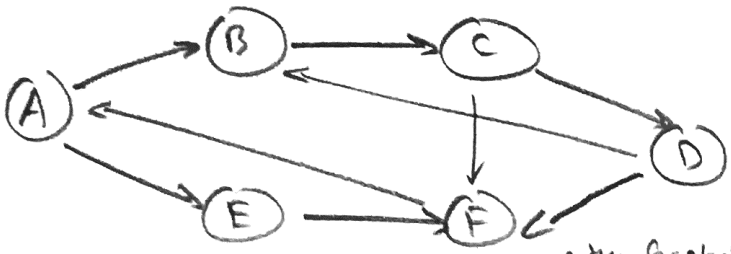

starting at A, we traverse all nodes (visit in alphabetical order if a tie).

pop off first element.

| Stack | ~~A~~ | E | ~~B~~ | ~~C~~ | ~~D~~ | ~~F~~ |
|---|---|---|---|---|---|---|
| Traversal | A | B | C | D | F | E |

(2) Breadth First Search : Queue.

Use a queue to implement BFS.




pop the first off like a queue

Queue | ~~A~~ ~~B~~ ~~E~~ ~~C~~ ~~F~~ ~~D~~

Traversal | A B E C F D

> Dijkstra big idea!
> Once a node is visited it is assumed that the distance given is the best (it will never visit that node again

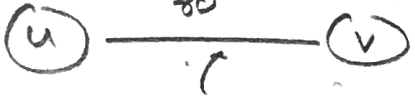
(3) Dijkstra's algorithm: Priority Queue.

It is recommended you also maintain a lookup table.

which you update based on this condition...

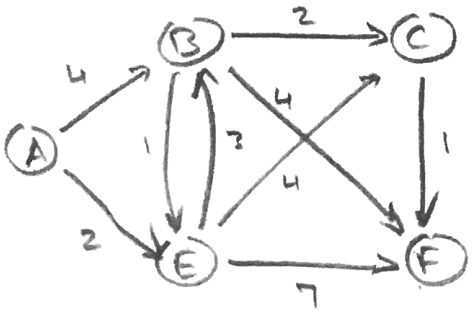
```
visit(u,v):
    if dist(u) + weight(u,v) < dist(v):
        dist(v) = dist(u) + weight(u,v)
```

basically says...




| cost | |
|---|---|
| u | 69. |
| v | ∞ |

if taking this path from u → v is better than the prev, then take that path...

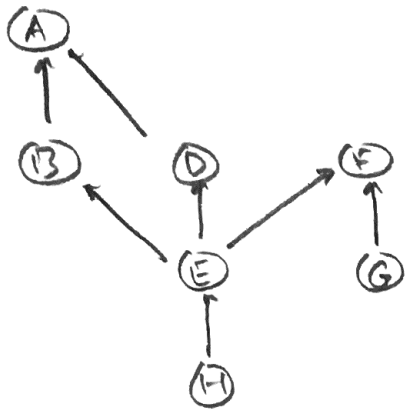



First start with all ∞ except start = 0

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| dist | 0 | ~~∞~~ | ~~∞~~ | ∞ | ~~∞~~ | ~~∞~~ |
| | | 4 | 6 | | 2 | ~~8~~ ~~8~~ 7 |

priority Q : ~~A~~ ~~E~~ ~~B~~ ~~C~~ ~~F~~

Graph ...




now start at G again.

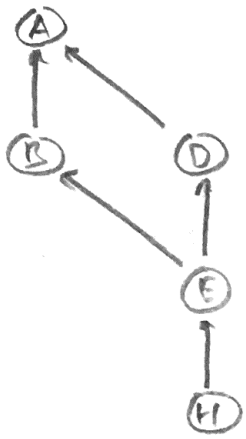
Stack : ~~G~~ ~~F~~

traversal : G (E) ✓ sink.

topological sort : C.

now added F and removed F. Makes G a sink vacuously.

topological sort C F G




stack | ~~H~~ ~~E~~ D ~~B~~ (A) ← no successors so no need to run entire DFS.

trav. | H E B (A)

end topos sort : C F G A B D E H.

and to summarize the algorithm is ...

```
Topos Sort(G):
    sort = new list.
    while graph still has nodes:
        u = sink(G)
        remove(u)
        add u to sort ...
    return sort.
```

to find sink.
reverse G edges
run DFS until
a node w/ no
neighbors is found.

Final distance table.

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | 4 | 6 | ∞ | 2 | 7 |

## Topological Sorting

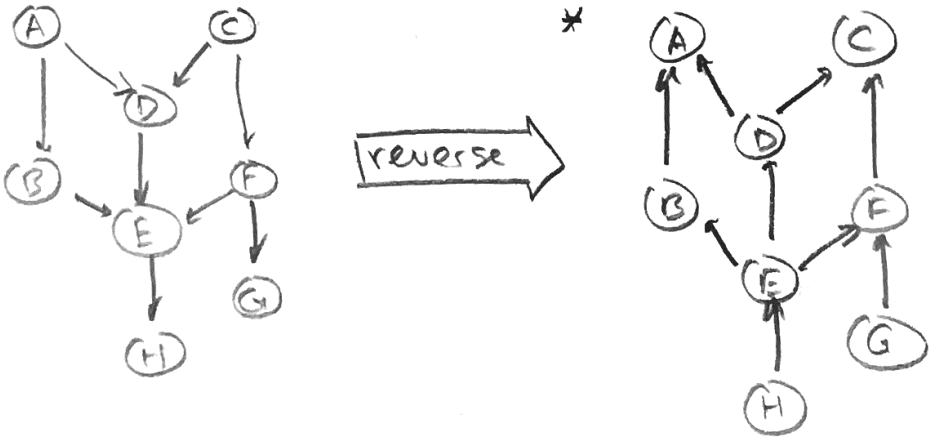
Given a DAG, find a linear order of nodes consistent with edges in a meaningful requires:

- for $v_0 \ldots v_k \rightarrow v_k$ is never reachable from $v_{k'}$ if $k' > k$

① Algorithm one...

```
topological (G, start node u):
    reverse edges in G.
    run dfs(G, u)
    ordering is a topos sort...
```

} except with caveats




- start at a source node and run DFS to find a sink.
- once sink is found add it to the traversal.
- Sink can be found by running DFS.
- remove sink and run again...

Example: starting with graph (*).

Stack | G F C

Traversal | G F C ← this is a sink.

- now remove C and add it to traversal..

# A* Search

So sometimes dijkstra's sucks ... why? Because say you want to search a path from Denver to NY.


Dijkstra will search to Seattle even though it's in the completely opposite direction. So, added a heuristic to guide it towards NY.

key idea!

- construct heuristic $H(v)$. that returns a value for how good a node is.
- order the priority queue by the sum of the distance plus the heuristic.

```
visit(u,v):
    if dist(u) + weight(u,v) + h(v) < dist(v):
        dist(v) = dist(u) + weight(u,v) + h(v).
```

note: you could also just order your PriorityQueue by dist(u) + weight(u,v) + h(v) and maintain a seperate travelCost ..

but there are a lot of details about heuristics.

## Heuristic Admissibility

We define an admissible heuristic as one that does not overestimate the goal ie if $h(v)$ is a heuristic function and $OPT(v)$ is the optimal cost.

$$\forall v \in \Omega : \quad h(v) \leq OPT(v).$$

$\Omega$: solution space.

For A*, the evaluation function is...

$$eval(v) = \underbrace{dist(v)}_{\text{current cost to get to } v} + \underbrace{h(v)}_{\text{heuristic for } v.}$$

How can this break things? Well remember Dijkstra works by assuming each node that is popped off the PQ has its optimal distance. But having a NON ADMISSABLE heuristic can fuck that up.

Ex: Goal go from Denver to NYC

- admissable heuristic! Use euclidean distance because it is always less than actual path.
- nonadmissable heuristic. for all $v \in$ Michigan $h(v) = 10{,}000$ makes sure no citys in michigan is processed by A* and when A* processes NYC (as a final step), the "optimal" path returned may not be optimal because no citys in Michigan was processed.

## Consistent Heuristic

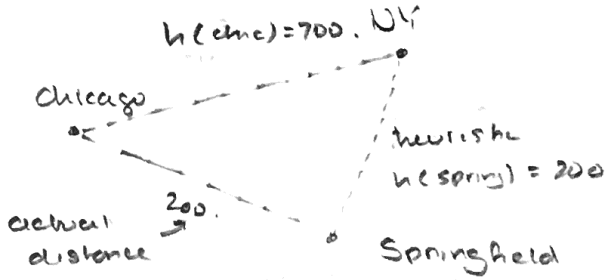
A heuristic $h(v)$ is consistent if the triangle inequality is satisfied namely, for two points in the search space, $u, v$.

$$\forall\, u, v \in \Omega \quad h(u) \leq h(v) + dist(u, v)$$

Intuitively that is it is faster to go through $u$ than a detour through $v$.

How can a non-consistent heuristic mess up our algorithm?

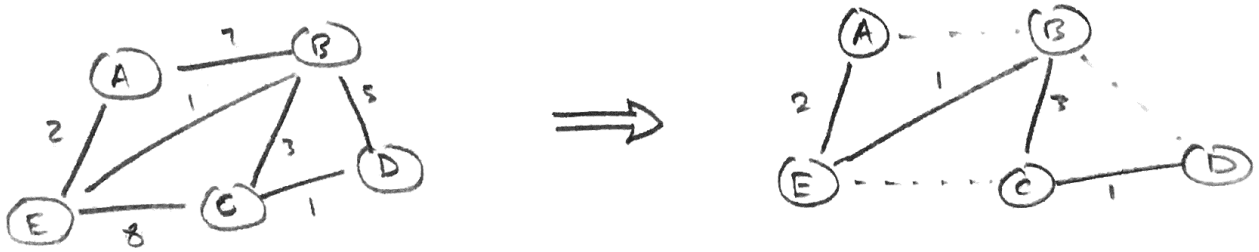
Consider ..




So by traveling through Springfield you cut out 500 miles from Chicago? (that's not right) so we require consistency..

## Minimum Spanning Trees

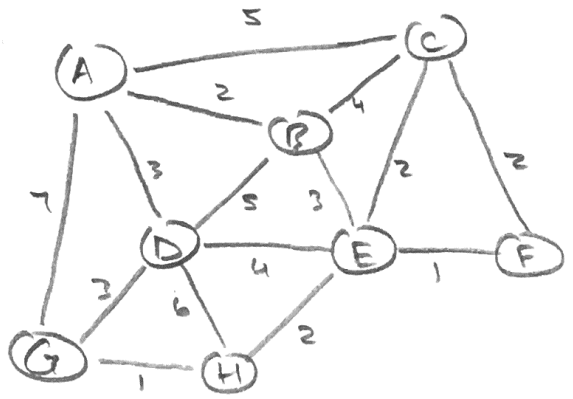
Problem statement: Given a graph $G$, construct a tree that spans all of nodes $V \in G$. such that the total weight of the tree is minimal. (MINIMAL TOTAL LENGTH)



There are two algorithms ...

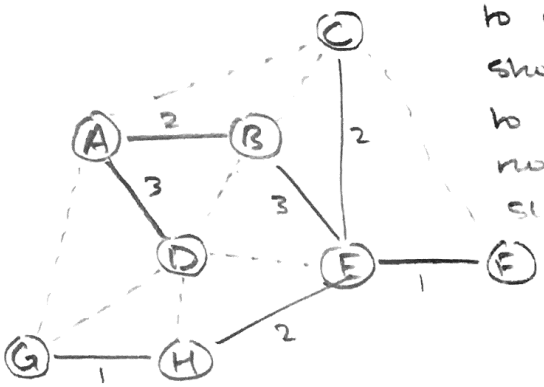
① Prim's algorithm.

- Add arbitrary starting node S.
- At each step
  - add the shortest edge connecting some node already in the tree to one that isn't in the tree.




② Krusleal's algorithm.

- at each iteration add an edge into the tree such that a cycle is not formed and edge weight is minimal.




* also it's smart to update the shortest distance to node in the node (like the slides ...) *

## Kruskal's Algorithm and Union Find

The actual algorithm actually goes like this.

- Place all nodes in its own subtree.
- * Pick two subtrees seperated by an edge of smallest length.
- add edge.
- repeat

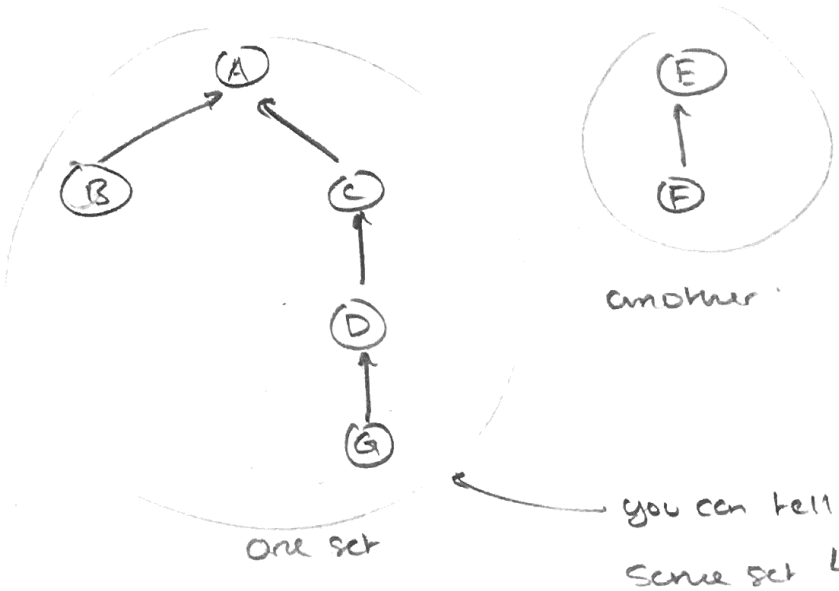
you can perform (*) by sorting all edges and then finding min edge that combines two seperate subtrees.

there's a faster way to combine subtrees... Have a bunch of sets of nodes and we want to do two operations...

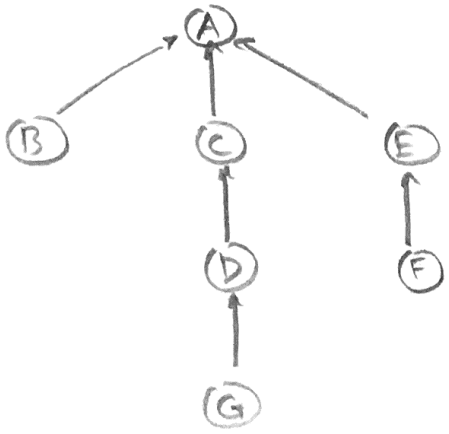
1) Find which set a node belongs to

2) Union two sets into one.

Solution: Let is choose to represent a set with an arbitrary representative...

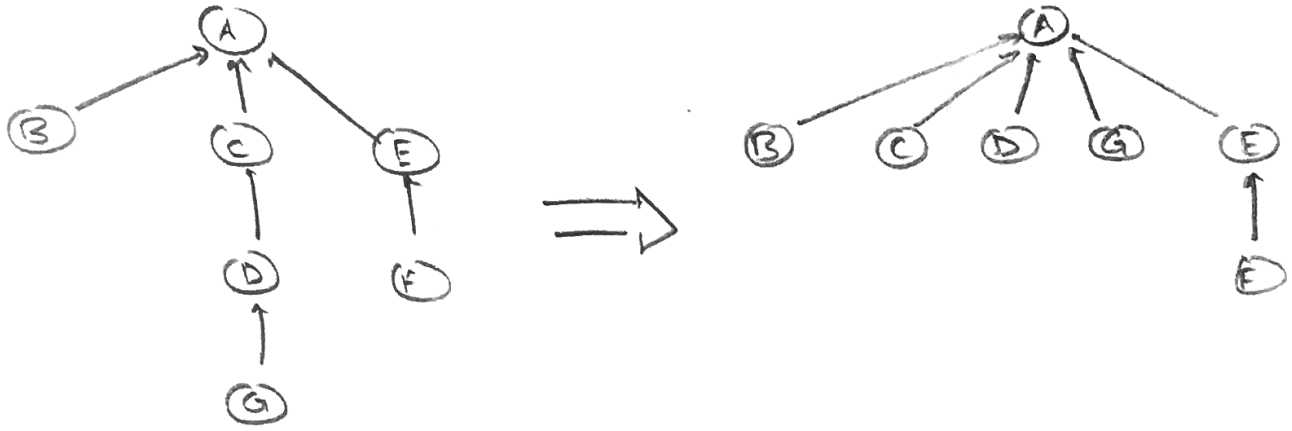



now given the root for two sets you can combine in constant time by having one point to another...



You can also use lazy computation to speed up finding. - -

by changing the parent pointer appropriately...




* Suppose you find G. you end up traversing D, C to get to A. *

* just have each node point to A to reduce the chance of having $\Omega(\log N)$ search *.

This leads to $\Omega(1)$ search time! (this is called path compression) actually cost is amurhized inverse ackerman.

## Time Bounds on Search

| | |
|---|---|
| Depth First Search | $O(\|E\|)$ if no repetition $O(b^d)$ with. |
| Breadth First Search | $O(\|E\|)$ |
| Dijkstra's Algorithm | $O(\|E\| + \|V\| \log \|V\|)$ ← using a fibonnaci heap. |
| A* search | $O(\|E\|)$ |
| prim's algorithm | $O(\|E\| + \|V\| \log \|V\|)$ ← using a fibonacci heap |
| kruskal's | $O(\|E\| \log \|E\|)$ or $O(\|E\| \log \|V\|)$. |
| topological sort | $O(\|V\| + \|E\|)$. |