Graph algorithms: Traversing the tree and beyond

Second in a 3-part series:

- Workshop 1: What is a graph and what can we do with it?
    - Available on the “WiDS Workshops” YouTube channel
- **Workshop 2: Graph algorithms: Traversing the tree and beyond**
- Workshop 3: Graphs in the real world
    - End of August

Julia Olivieri

jolivier@stanford.edu

Code for today:

https://github.com/juliaolivieri/WiDS_graph_algorithms

What is a graph?

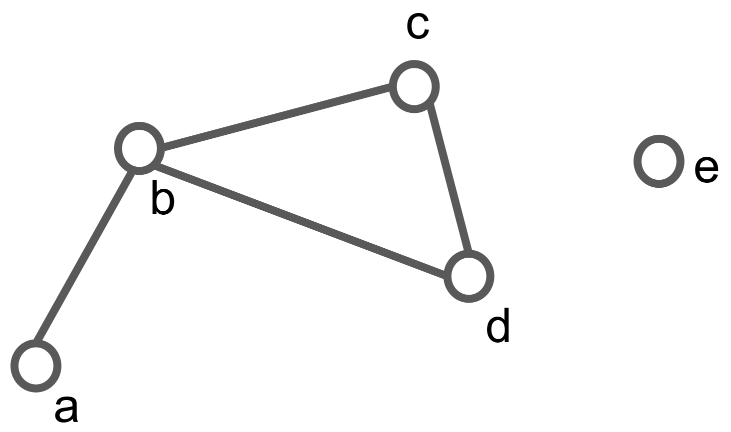
A graph **G** is a pair of sets (**V**, **E**) satisfying the following two conditions:

1. **V** is finite and non-empty
2. Each element of **E** is a 2-element subset of **V**

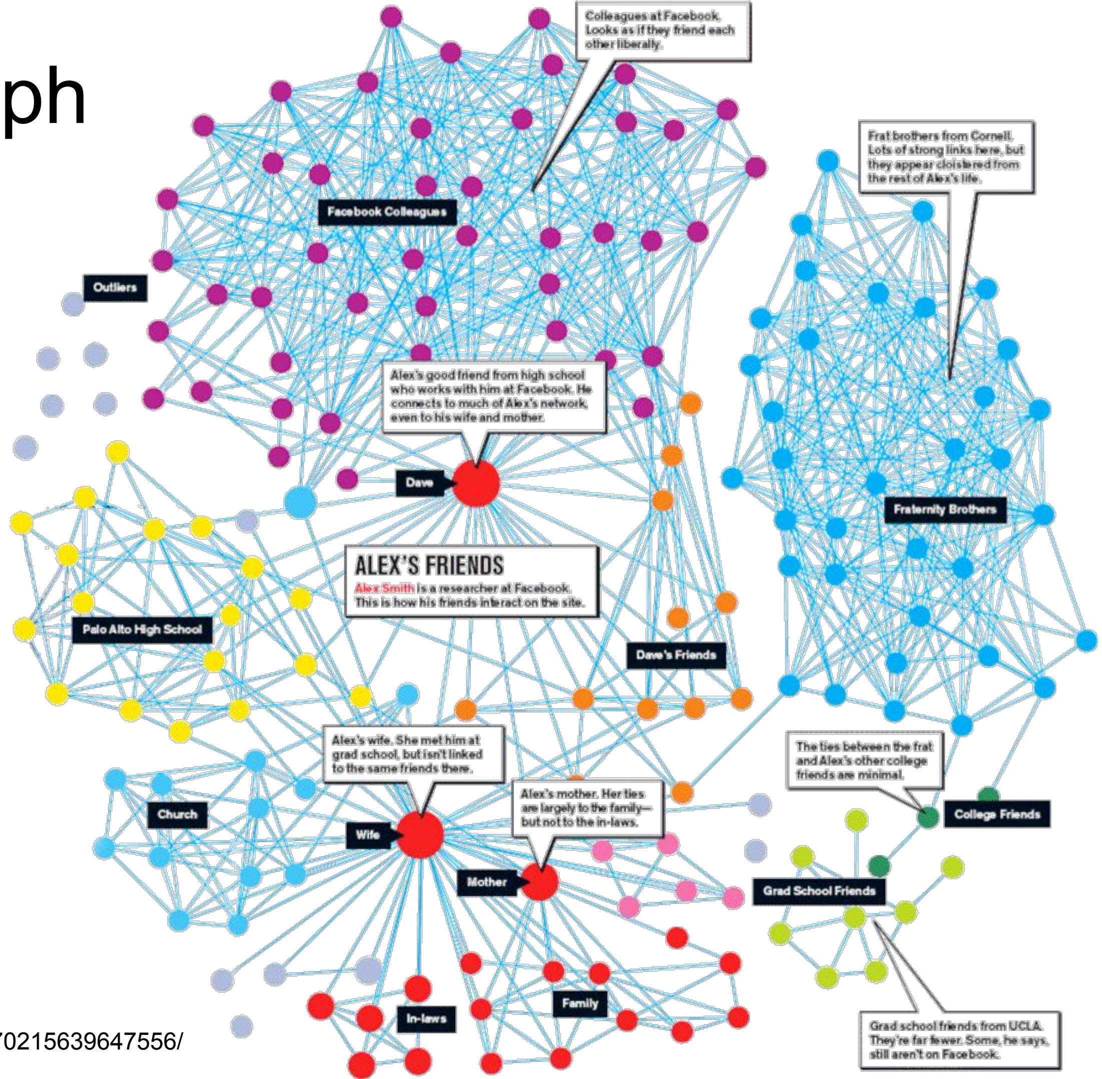
Example:

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

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

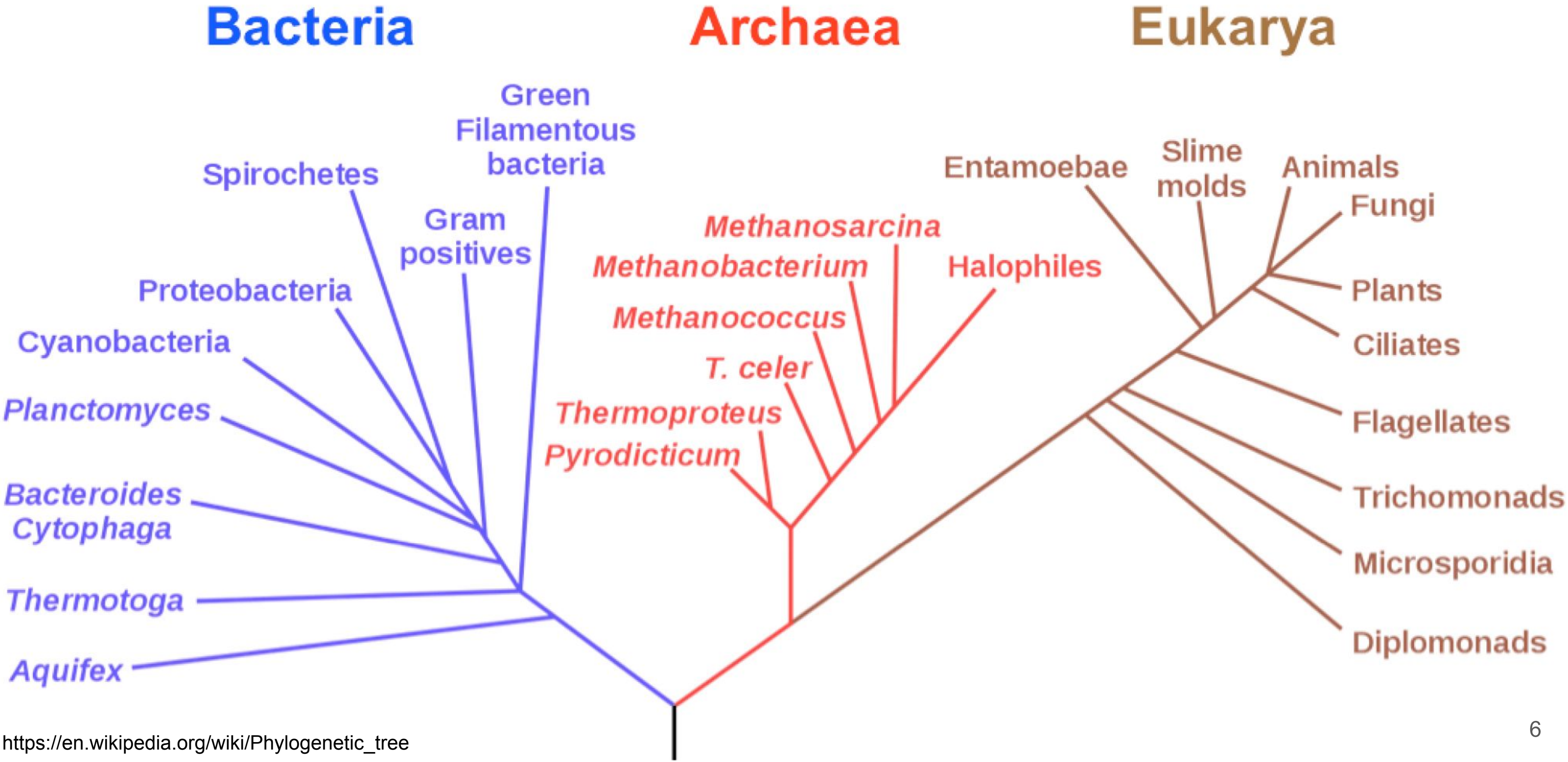


Social graph



https://www.pinterest.com/pin/490470215639647556/

Phylogenetic Tree



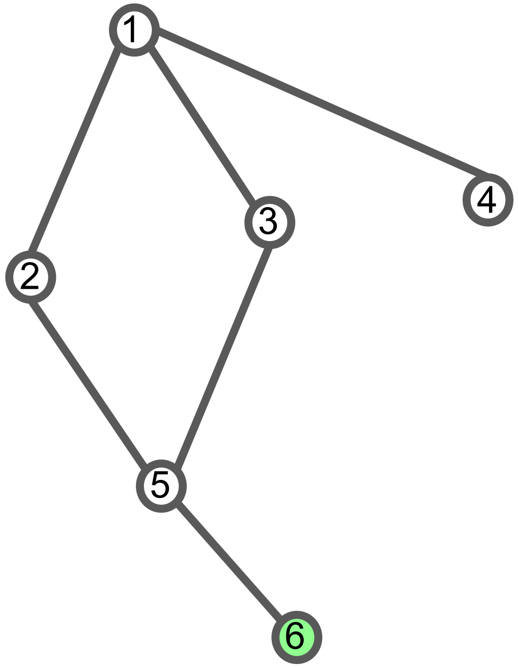
https://en.wikipedia.org/wiki/Phylogenetic_tree

Map

https://www.benchmarkmaps.com
/product/california-road-map

Graph traversal: How do we search for something in a graph?

We want to find the green vertex. What do we do?



Data structures to store graphs

Adjacency list/adjacency dictionary: For each vertex, we store a list of all the neighbors of that vertex

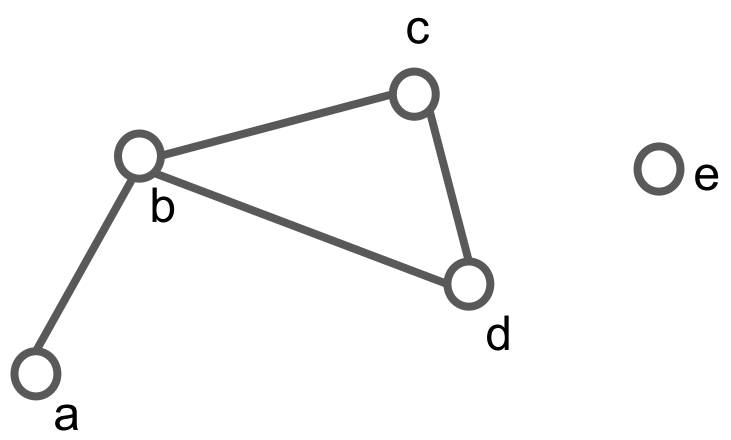
adj = { a : [b],

b : [a, c, d],

c : [b, d]

d : [b, c]

e : [] }

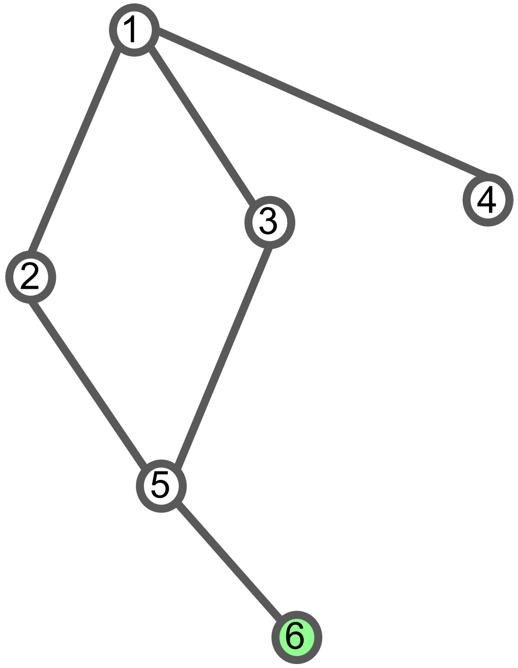


Two main methods of graph traversal

- **Breadth First Search (BFS)**
- Depth First Search (DFS)

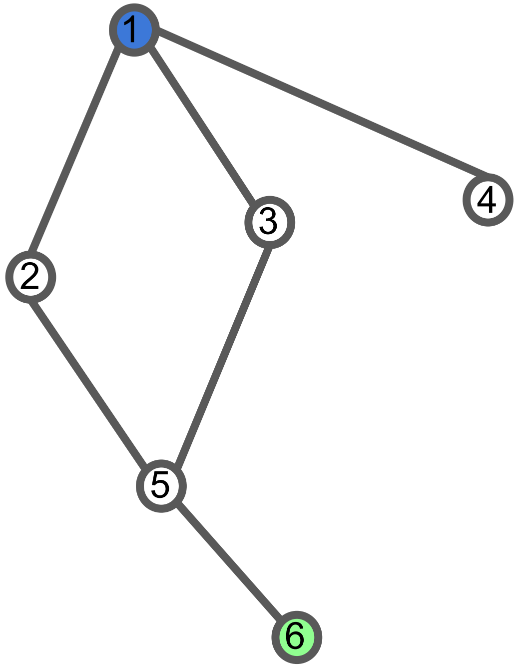
Breadth First Search (BFS)

- Search every neighbor of the starting vertex



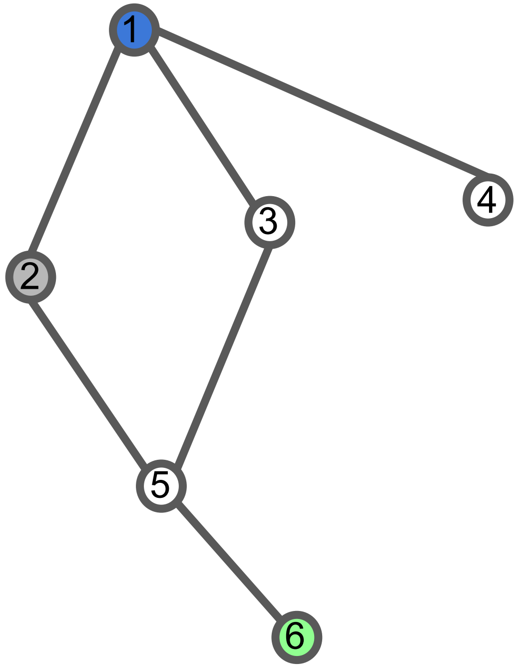
Breadth First Search (BFS)

- Search every neighbor of the starting vertex



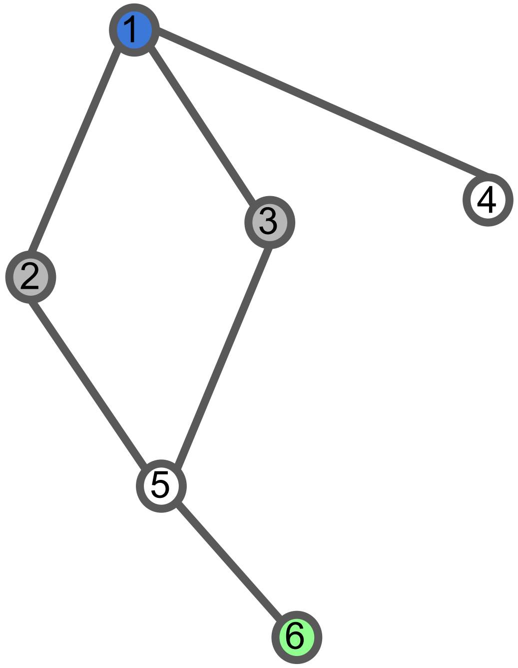
Breadth First Search (BFS)

- Search every neighbor of the starting vertex



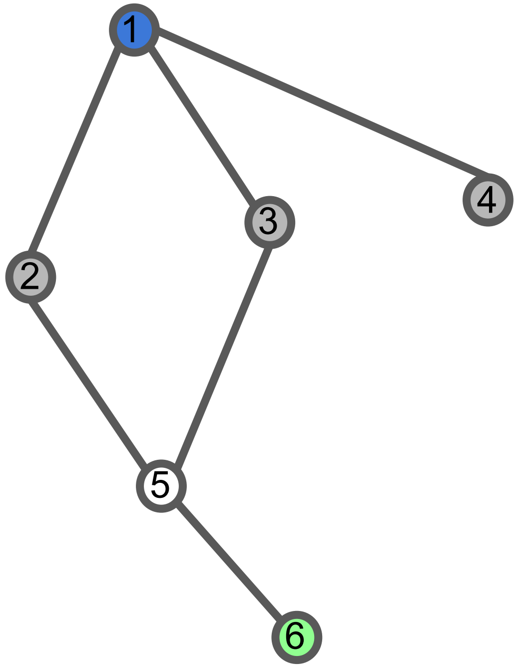
Breadth First Search (BFS)

- Search every neighbor of the starting vertex



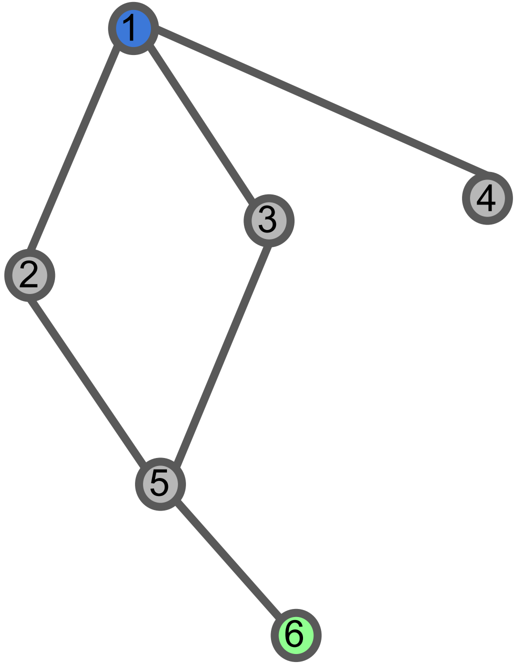
Breadth First Search (BFS)

- Search every neighbor of the starting vertex



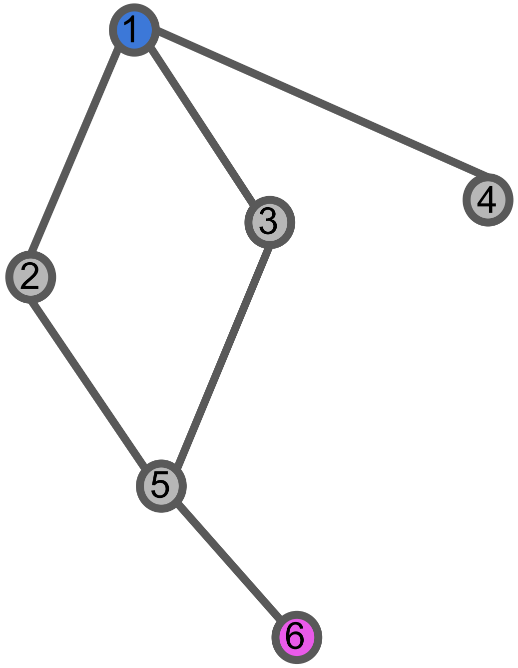
Breadth First Search (BFS)

- Search every neighbor of the starting vertex
- Then search every neighbor of the neighbors



Breadth First Search (BFS)

- Search every neighbor of the starting vertex
- Then search every neighbor of the neighbors
- Continue searching outwards “in layers”
- BFS is useful when solutions are expected to be close to the starting vertex
  - Finding someone with the same job as you in a social network

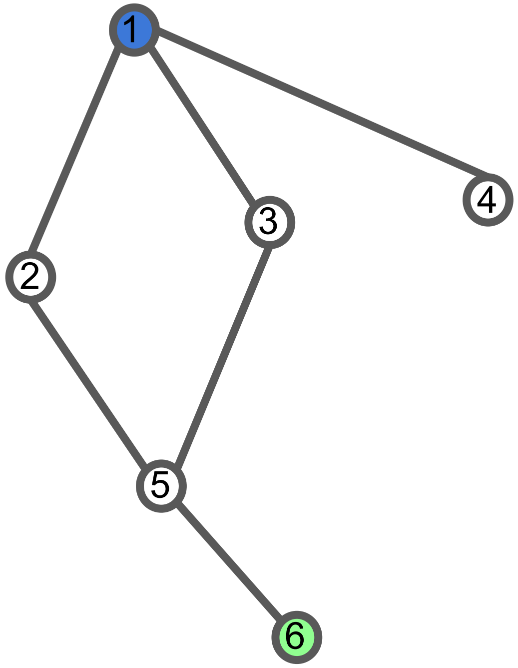


Two main methods of graph traversal

- Breadth First Search (BFS)
- **Depth First Search (DFS)**

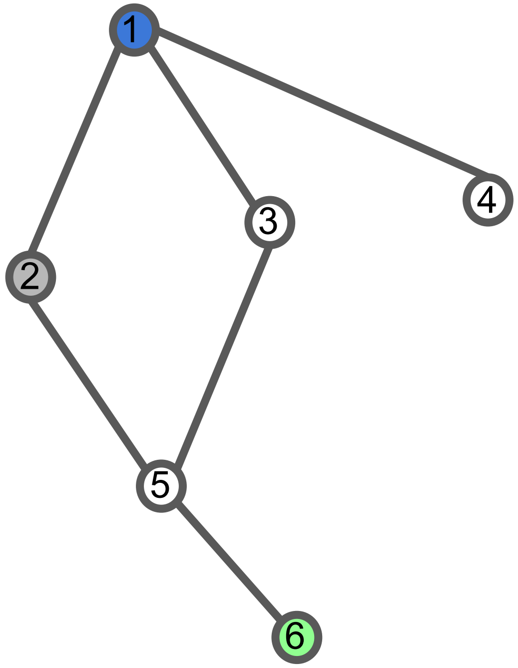
Depth First Search (DFS)

- Search a neighbor of the first vertex



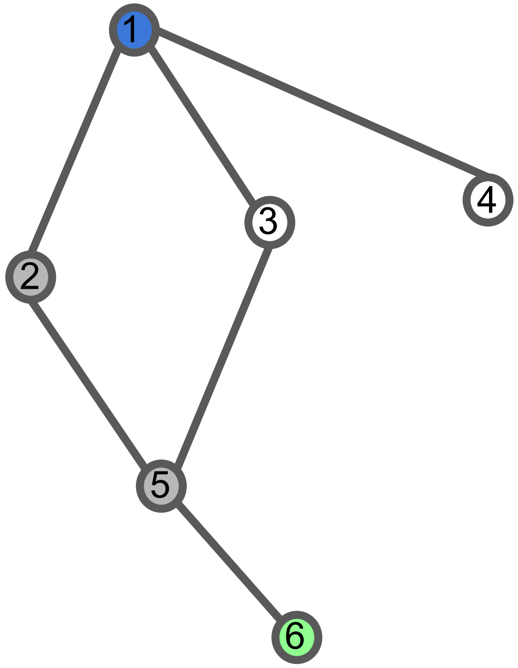
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex



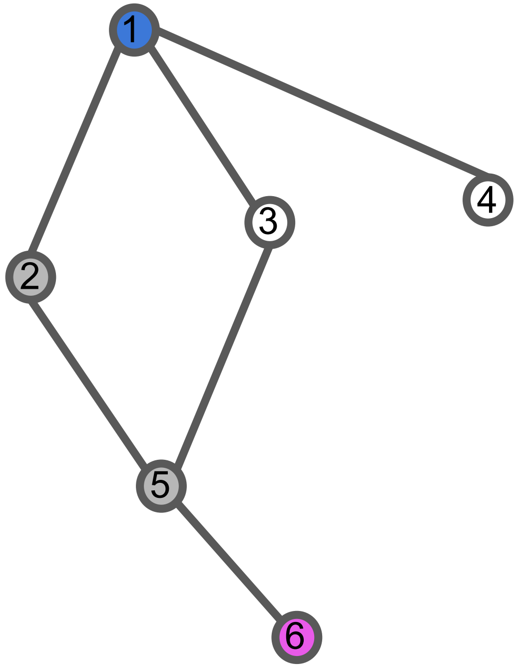
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors



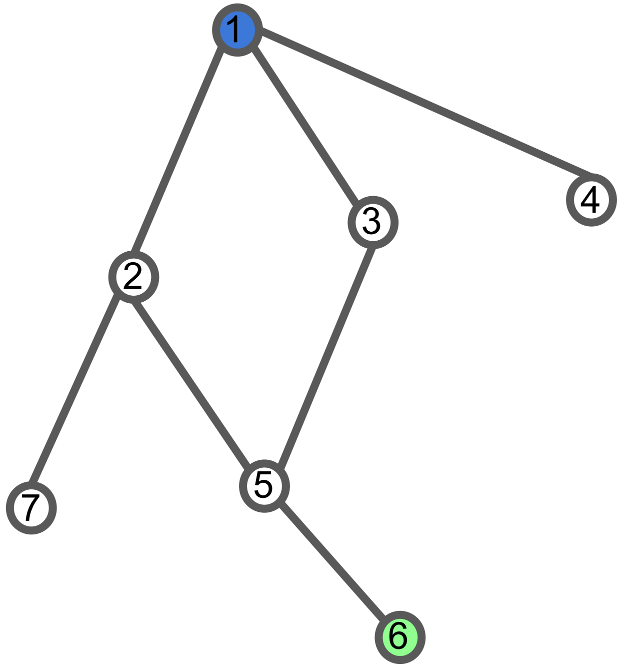
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors



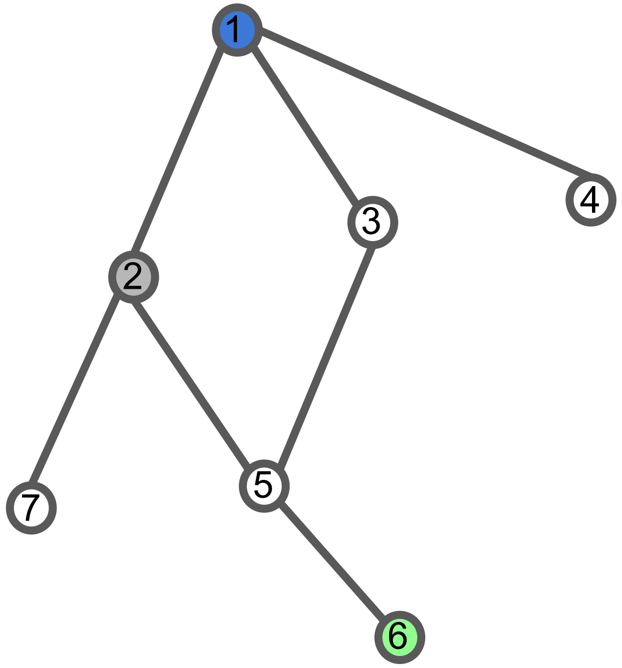
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors



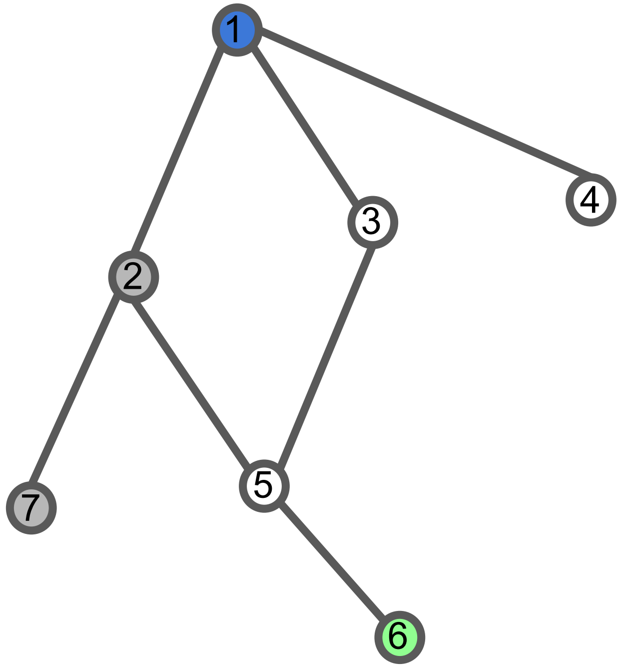
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors



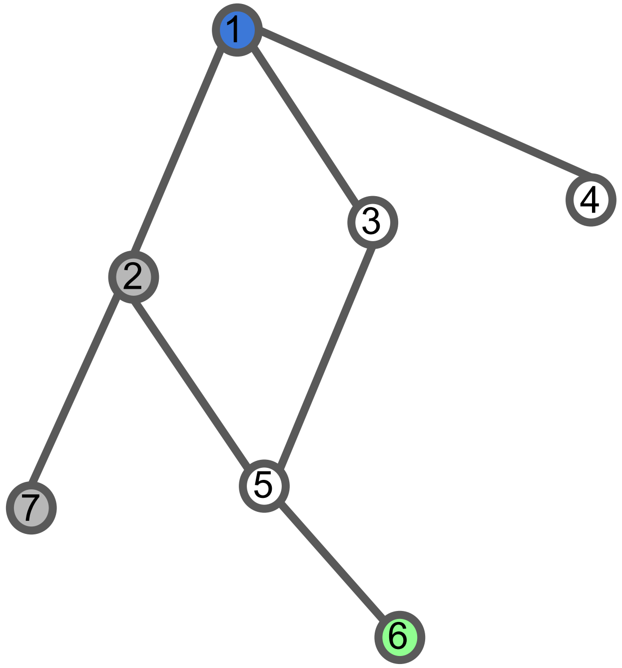
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors



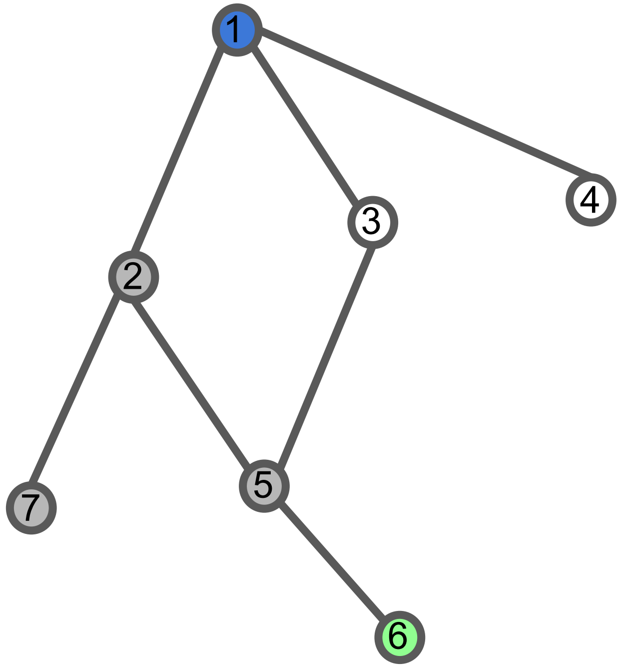
Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors
- Backtrack and repeat



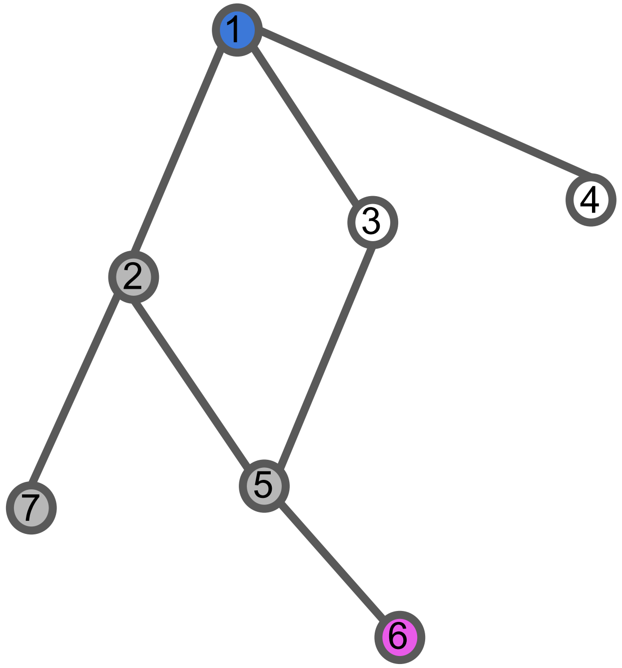
Depth First Search (DFS)

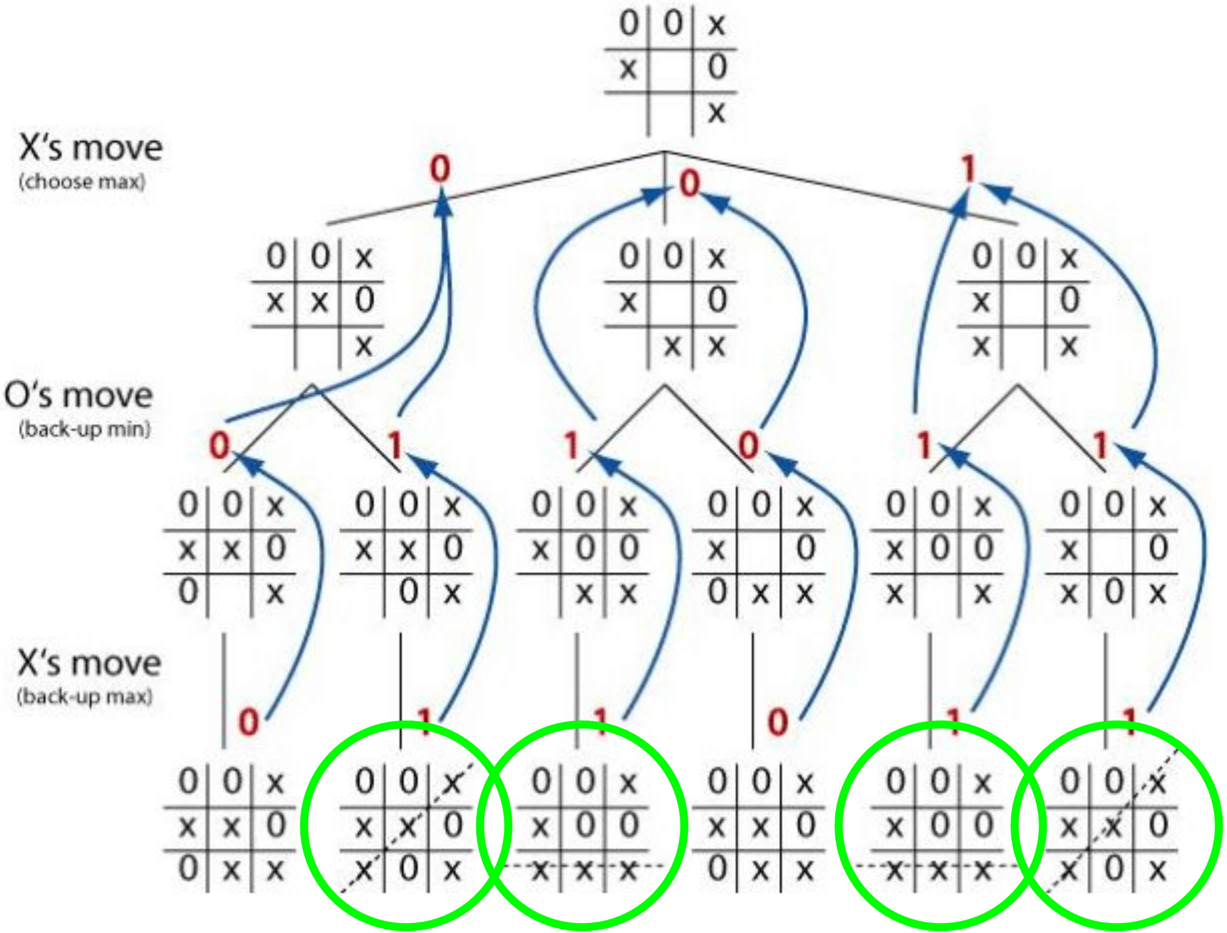
- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors
- Backtrack and repeat



Depth First Search (DFS)

- Search a neighbor of the first vertex
- Search an unsearched neighbor of the current vertex
- Continue until there are no unsearched neighbors
- Backtrack and repeat
- DFS is useful when solutions are expected to be far from the starting vertex
    - Finding winning solutions to a game

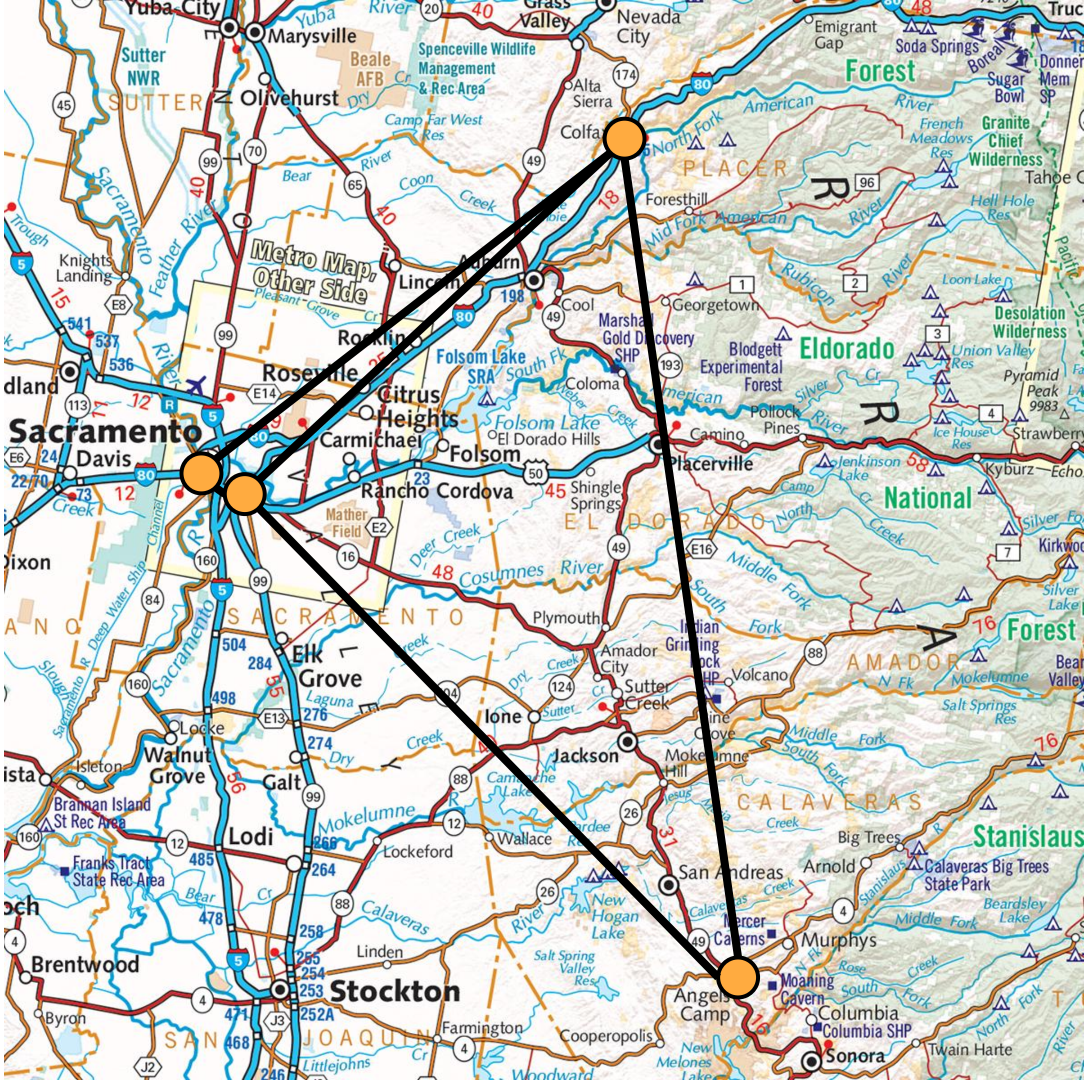




https://www.researchgate.net/figure/Game-tree-for-Tic-Tac-Toe-game-using-MiniMax-algorithm_fig1_262672371

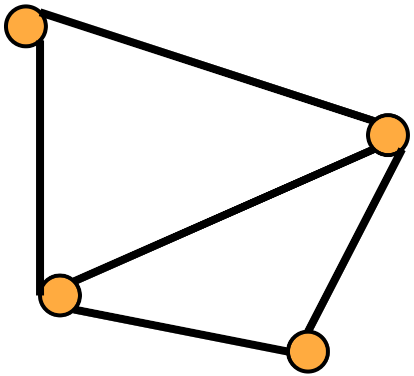
Breadth First Search	**Depth First Search**
Good for when solutions are close to the start	Good for when solutions are far from the start
Time complexity: O(E + V)	Time complexity: O(E + V)
Space complexity: O(V)	Space complexity: O(V)
Finds the closest solution to the starting point	Doesn't necessarily find the closest solution

Map



https://www.benchmarkmaps.com
/product/california-road-map

Map

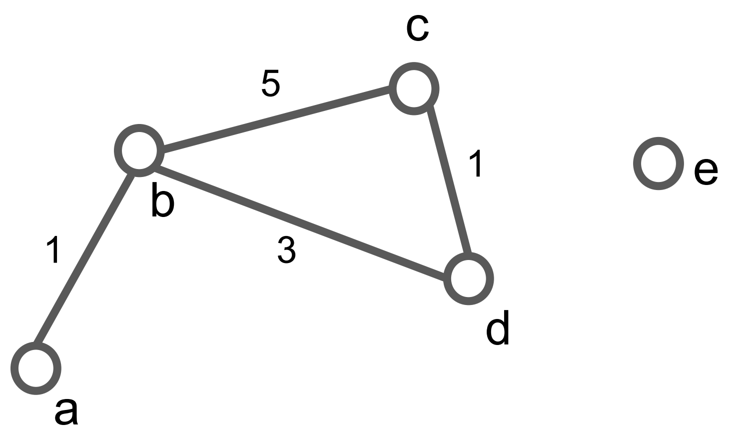


Weighted graph

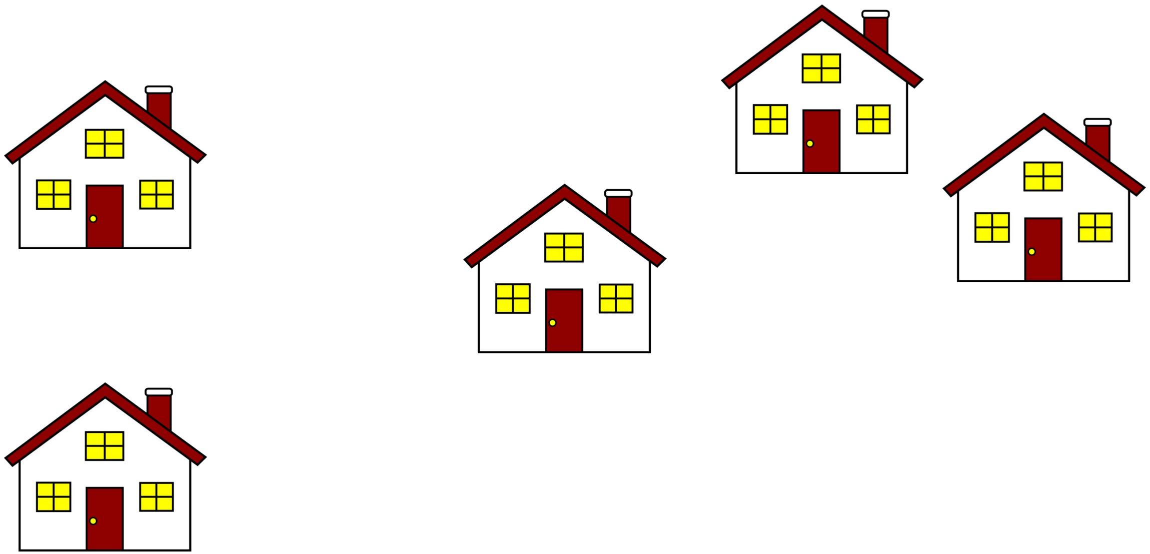
V = {a, b, c, d, e}

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

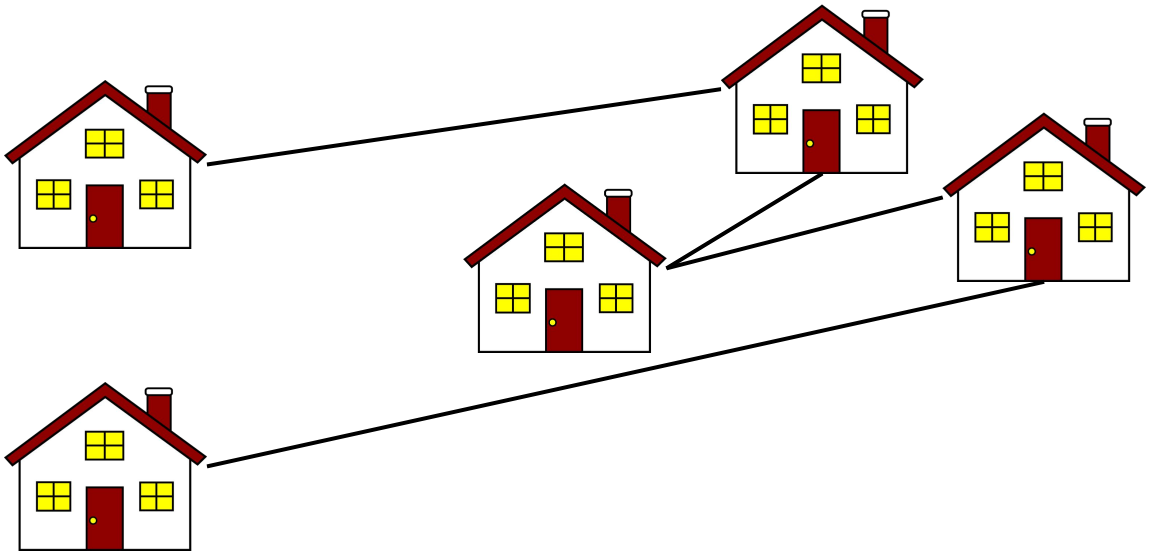
W = {(a, b) : 1, (b, d) : 3, (b, c) : 5, (c, d) : 1}



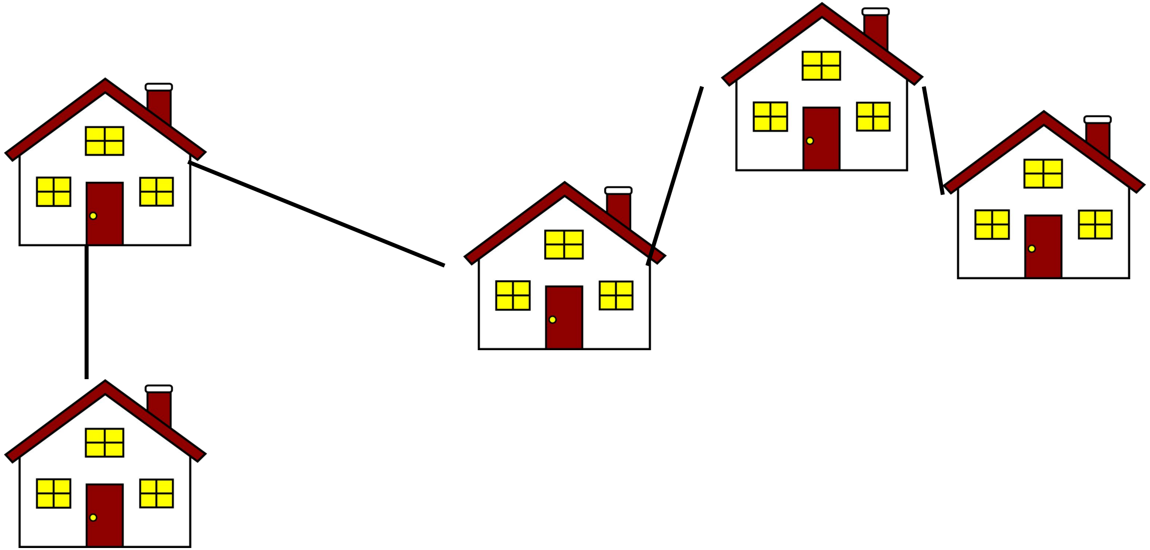
Minimum Spanning Tree: How do we find the tree connecting all the vertices with the smallest total weight?



Minimum Spanning Tree: How do we find the tree connecting all the vertices with the smallest total weight?



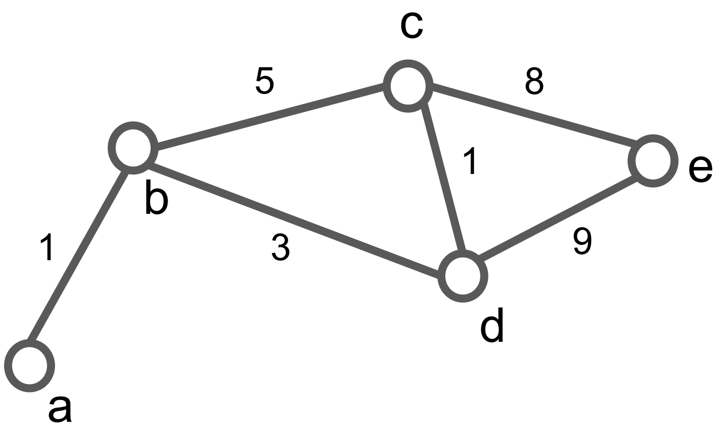
Minimum Spanning Tree: How do we find the tree connecting all the vertices with the smallest total weight?



Minimum spanning tree: Subset of edges of a connected weighted graph that **connects all the vertices** together **without any cycles** and with the **minimum** possible total **edge weight**

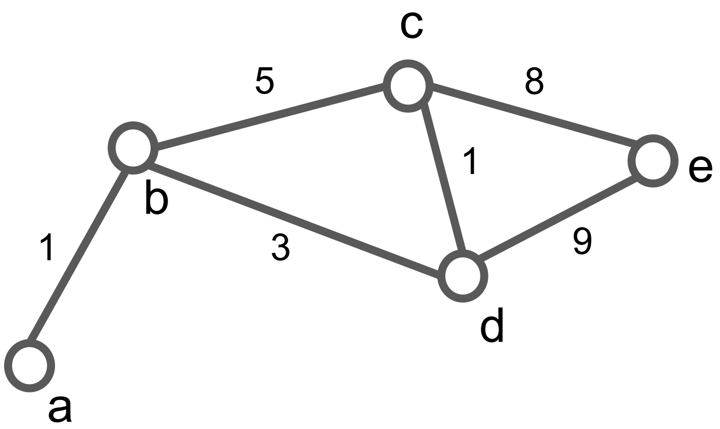
Kruskal’s Algorithm Example

1. Test if the graph is connected



Kruskal's Algorithm Example

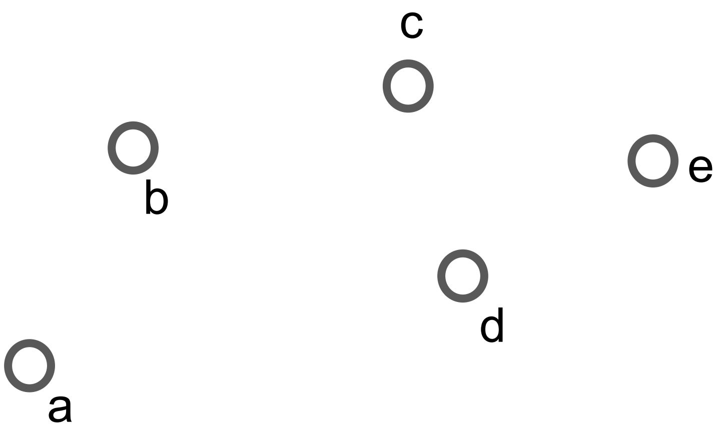
2. Order the edges from smallest to largest weight



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal's Algorithm Example

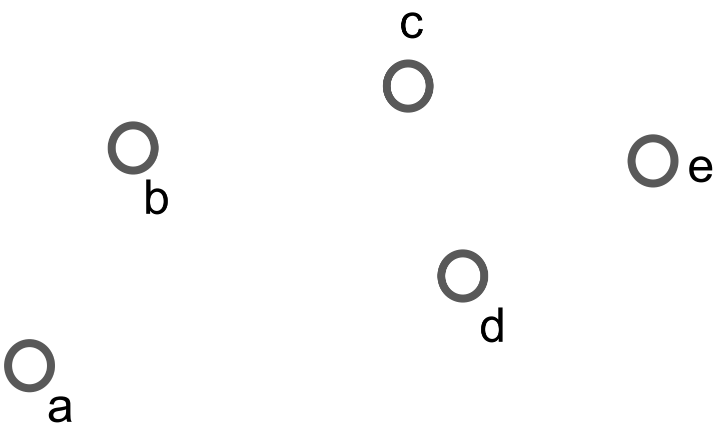
3. Start a graph with all the vertices but no edges, *T*



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal’s Algorithm Example

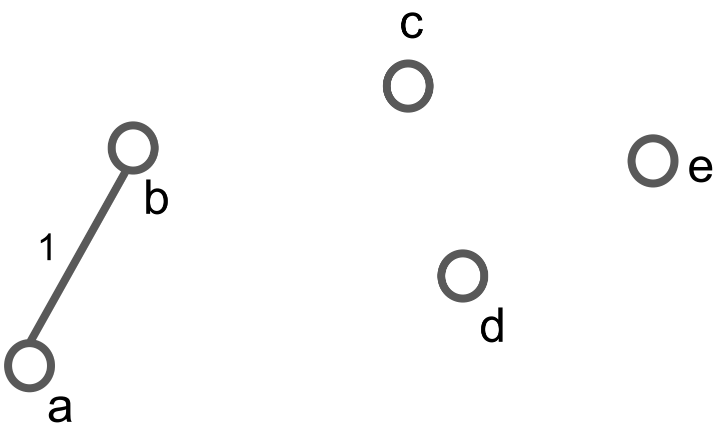
4. From the lowest weight to the highest weight edge:
   a. Try adding the edge to *T*
   b. If this causes a cycle, remove the edge
   c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal's Algorithm Example

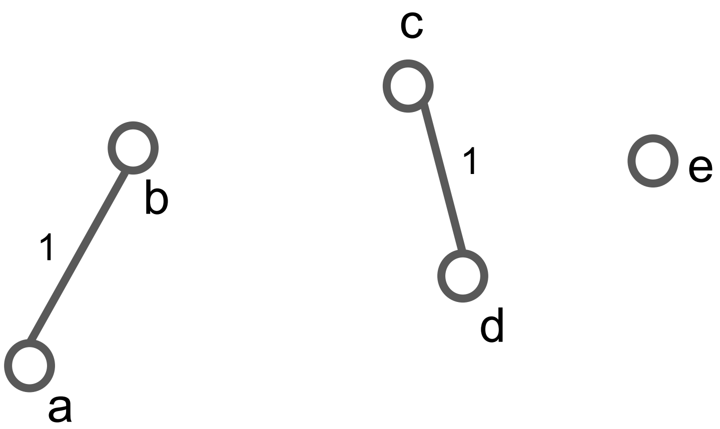
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. Else, if *T* is now connected, return *T*



(a,b)	**1**
(c,d)	1
(b,d)	3
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal’s Algorithm Example

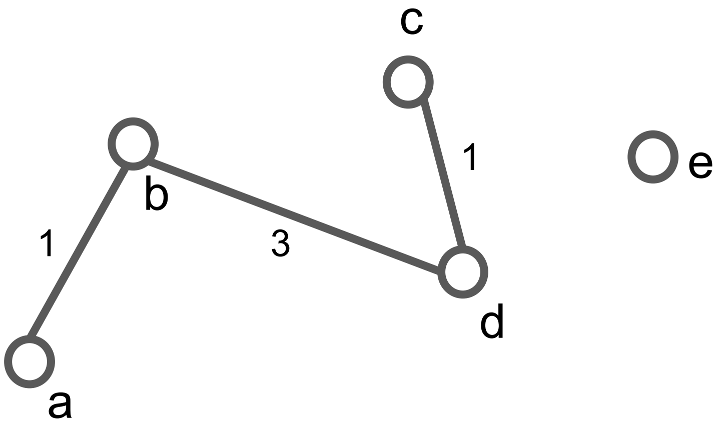
4. From the lowest weight to the highest weight edge:
   a. Try adding the edge to *T*
   b. If this causes a cycle, remove the edge
   c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	**1**
(b,d)	3
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal's Algorithm Example

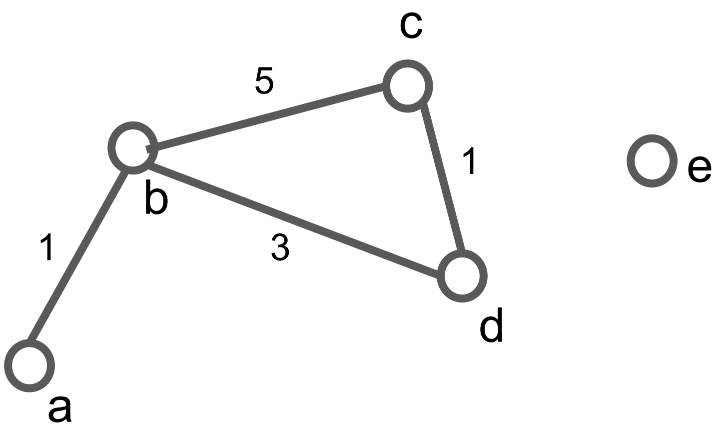
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	1
(b,d)	**3**
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal's Algorithm Example

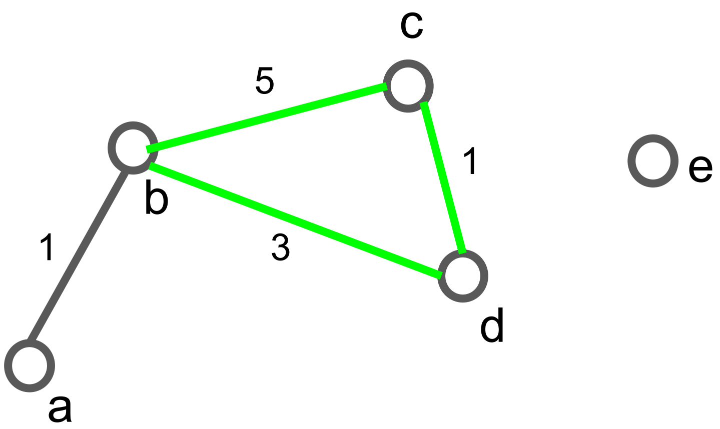
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	**5**
(c,e)	8
(d,e)	9

Kruskal’s Algorithm Example

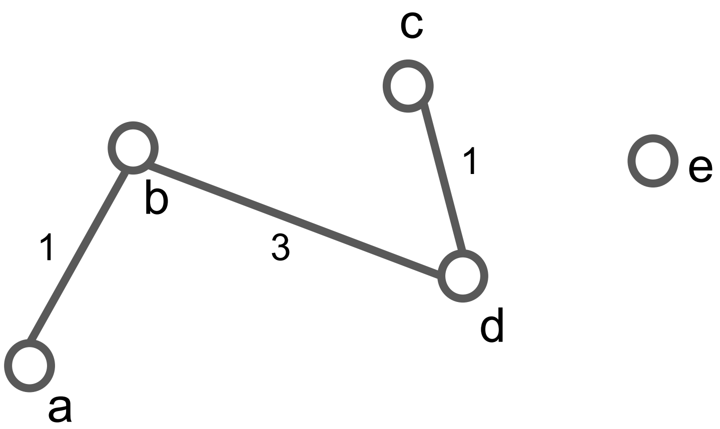
4. From the lowest weight to the highest weight edge:
   a. Try adding the edge to *T*
   b. If this causes a cycle, remove the edge
   c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	**5**
(c,e)	8
(d,e)	9

Kruskal's Algorithm Example

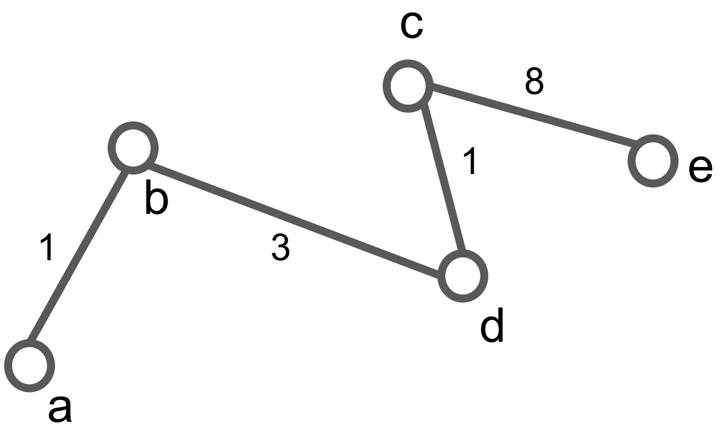
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	5
(c,e)	8
(d,e)	9

Kruskal's Algorithm Example

4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. Else, if *T* is now connected, return *T*



(a,b)	1
(c,d)	1
(b,d)	3
(b,c)	5
(c,e)	**8**
(d,e)	9

MST weight: 13

Kruskal’s Algorithm

1. Test if the graph is connected
2. Order the edges from smallest to largest weight
3. Start a graph with all the vertices but no edges, *T*
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. Else, if *T* is now connected, return *T*

Proof: https://en.wikipedia.org/wiki/Kruskal's_algorithm

Kruskal’s Algorithm

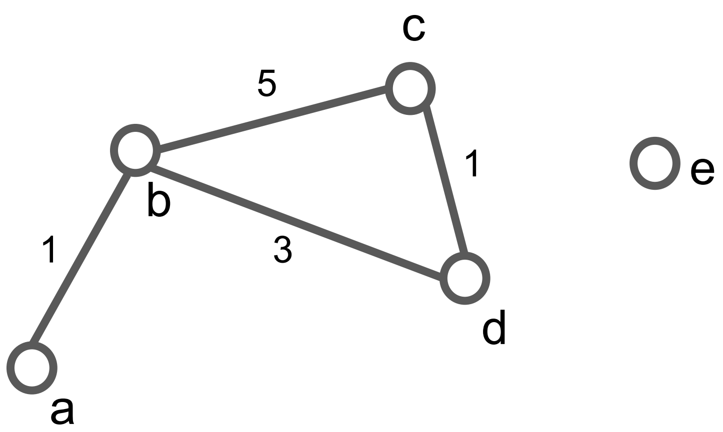
1. **Test if the graph is connected**
2. Order the edges from smallest to largest weight
3. Start a graph with all the vertices but no edges, *T*
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. If this causes a cycle, remove the edge
    c. **Else, if *T* is now connected, return *T***

adj = { a : [b],
b : [a, c, d],
c : [b, d]
d : [b, c] }

Proof: https://en.wikipedia.org/wiki/Kruskal's_algorithm

How do we test if a graph is connected?

1. Run DFS or BFS from any vertex
2. If it visits all the vertices, the graph is connected
3. If not, the graph is not connected



Kruskal’s Algorithm

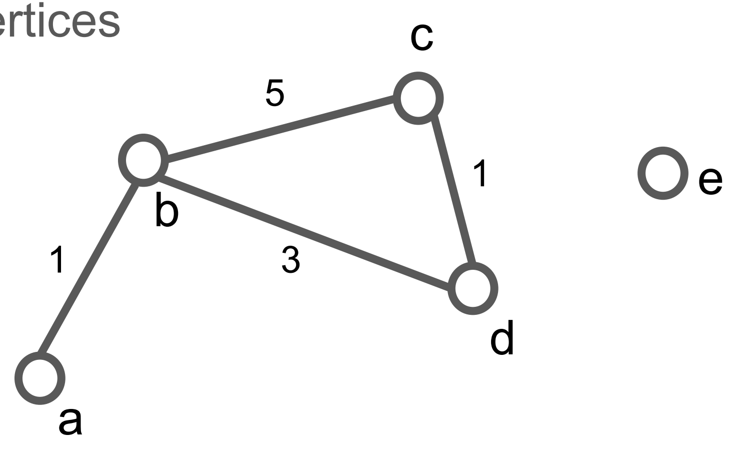
1. Test if the graph is connected
2. Order the edges from smallest to largest weight
3. Start a graph with all the vertices but no edges, *T*
4. From the lowest weight to the highest weight edge:
    a. Try adding the edge to *T*
    b. **If this causes a cycle, remove the edge**
    c. Else, if *T* is now connected, return *T*

Proof: https://en.wikipedia.org/wiki/Kruskal's_algorithm

How do we test if a graph contains a cycle?

1. Run DFS or BFS from any vertex
2. Using the list of visited vertices, count the number of edges between those vertices
3. If the *number of edges* > *number of vertices* – 1, the graph contains a cycle
4. Otherwise repeat for any unvisited vertices

There is another famous cycle-finding algorithm using DFS

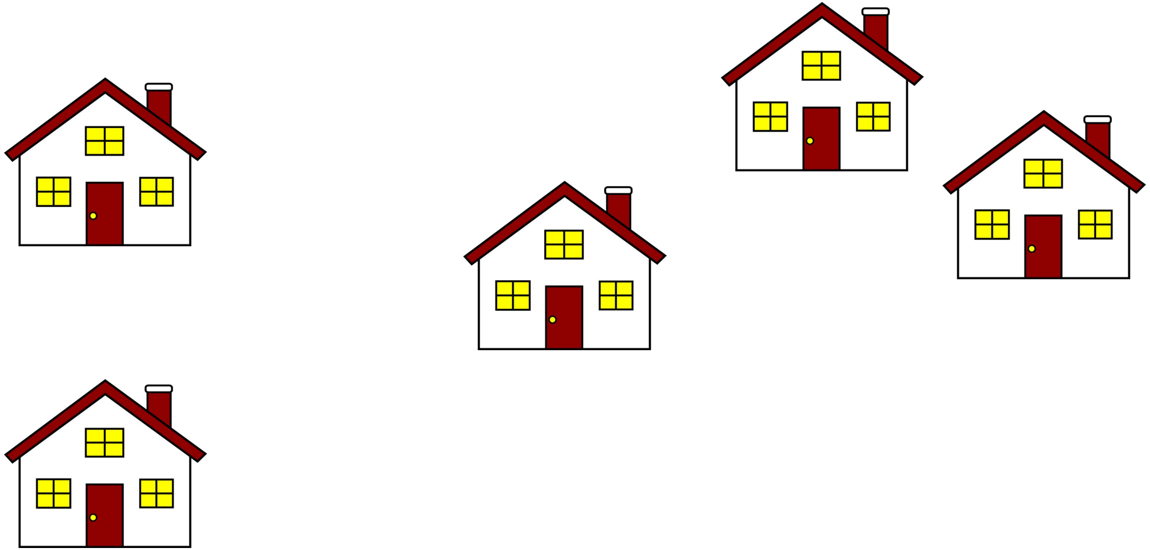


Kruskal’s Algorithm Complexity:

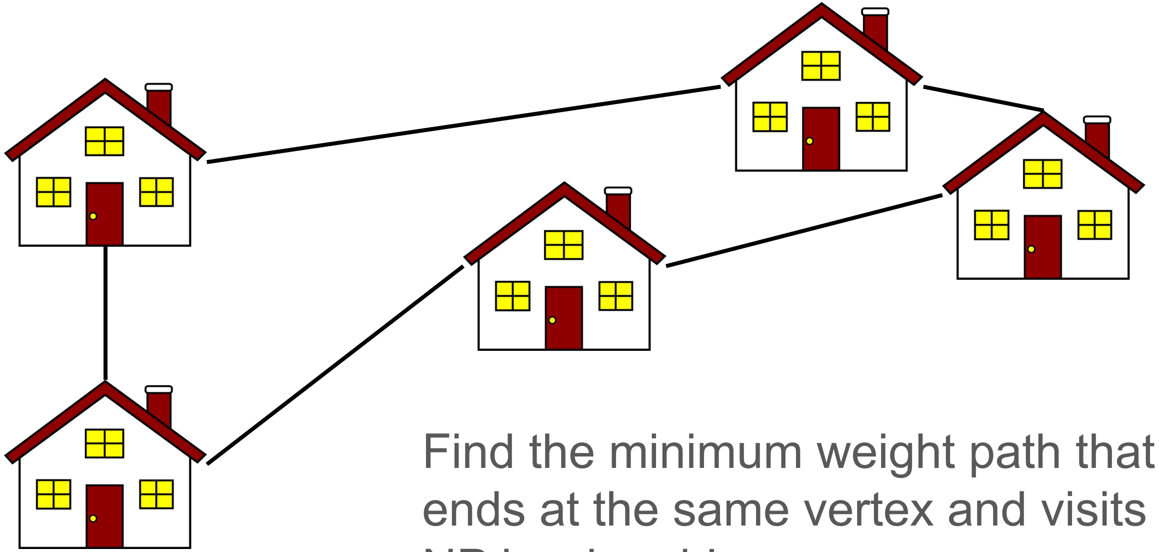
Time complexity:
O(E log E)

Space complexity:
O(E + V)

Traveling Salesman Problem



Traveling Salesman Problem



Find the minimum weight path that starts and ends at the same vertex and visits every vertex
NP hard problem
It has not been proved whether a solution $< O(1.9999^V)$ exists

Approximation algorithms

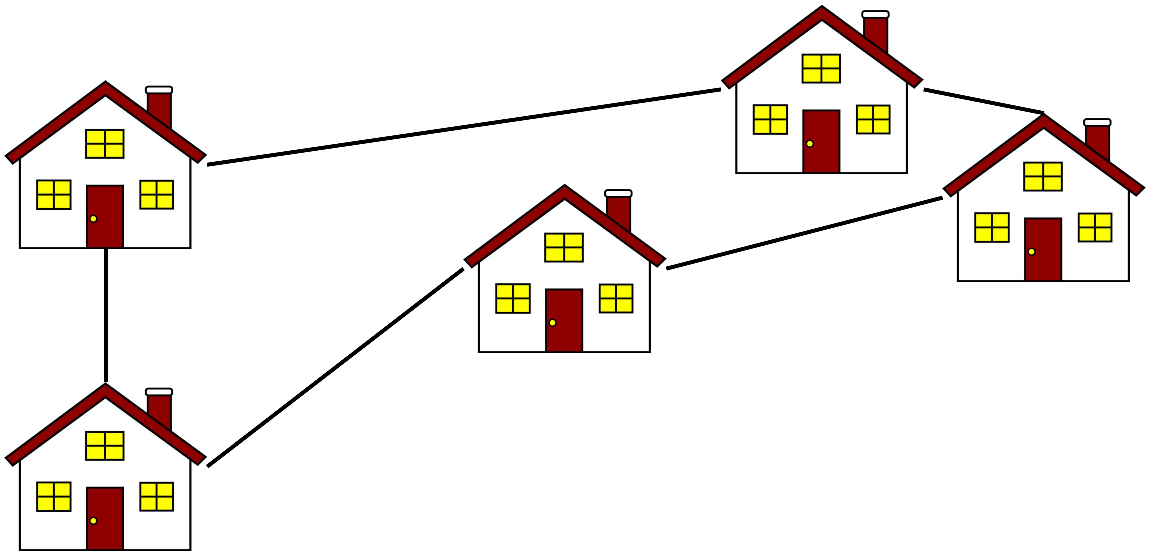
When a problem is NP hard, approximation algorithms can be used to get solutions that are provably **only a factor away from the optimal solution**

Let the optimal value be *OPT*. If we are trying to find a minimum, A ***k*-approximation algorithm** is an algorithm that returns a value *APRX* such that

$$OPT \leq APRX \leq k\ OPT$$

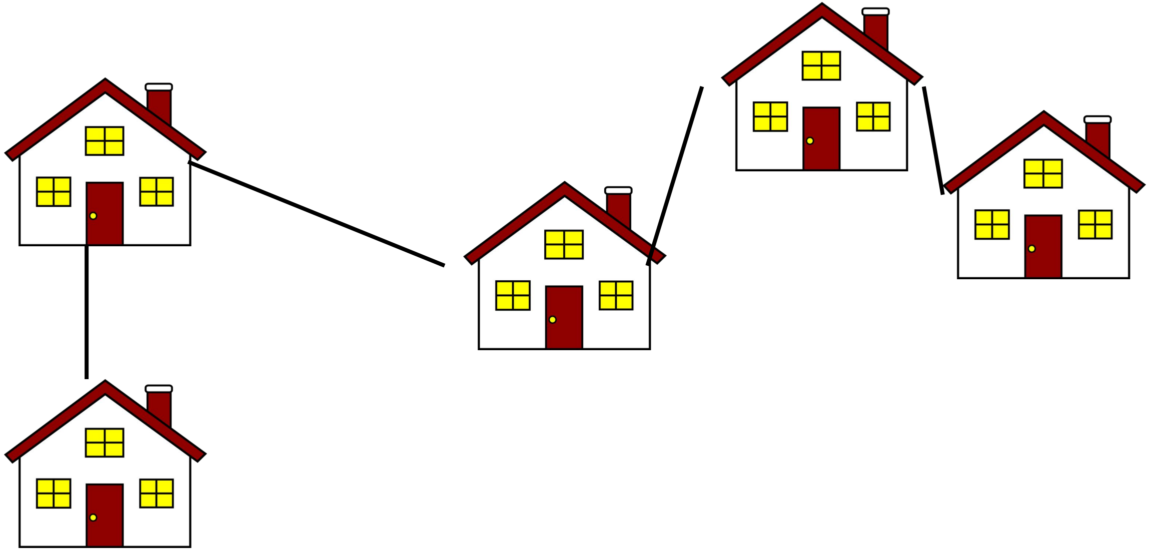
Traveling Salesman Problem

Deleting an edge from the TSP circuit creates a spanning tree



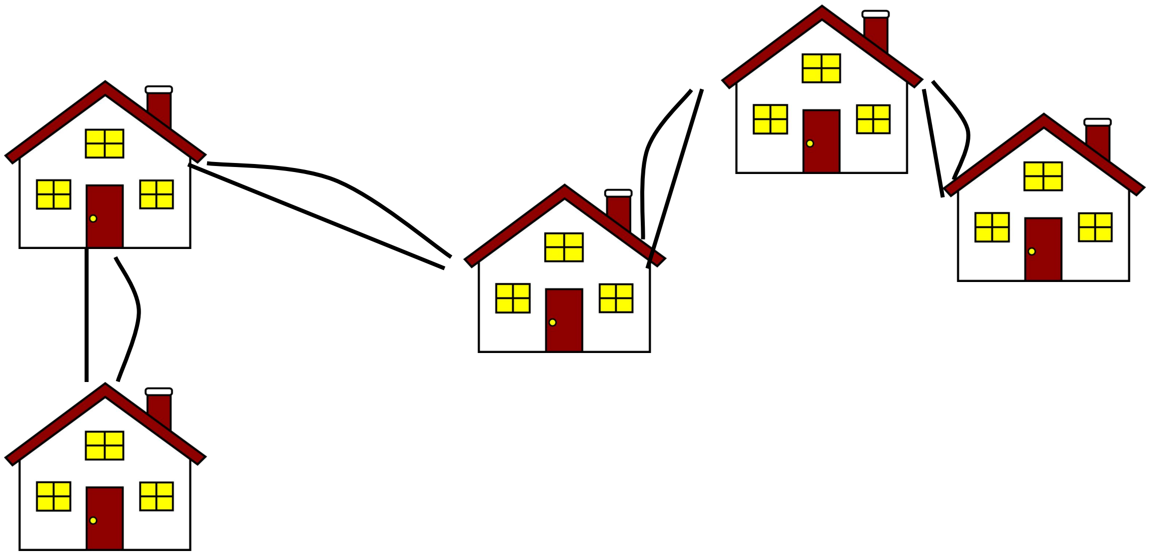
Traveling Salesman Problem

The length of the MST must be < *OPT*



Traveling Salesman Problem

The length of the MST must be < *OPT*



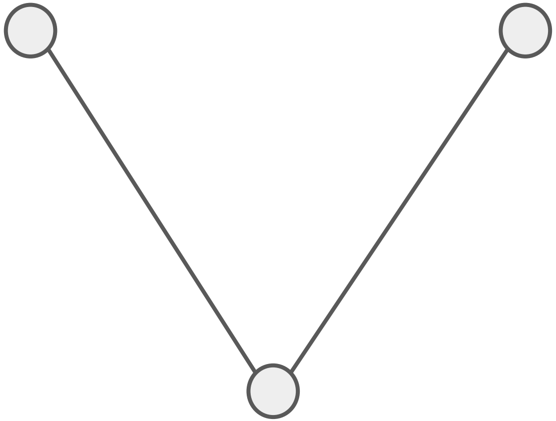
Then APRX ≤ 2 *OPT*, so this is a 2-approximation algorithm
The best known approximation algorithm has $k \approx 1.5$

Steiner Tree Problem

Given a weighted graph and a subset of the vertices, find a tree of minimum weight that contains all vertices in the subset (though it can contain more)

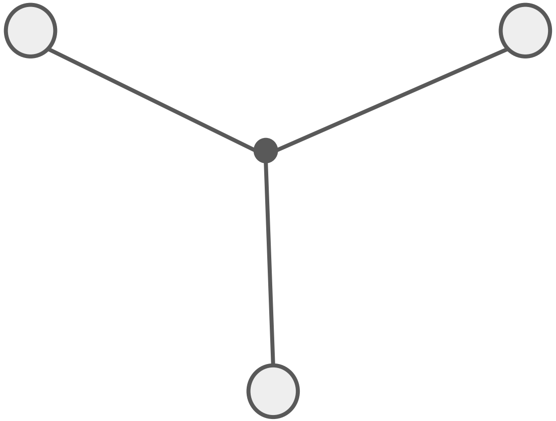
Steiner Tree Problem

Given a weighted graph and a subset of the vertices, find a tree of minimum weight that contains all vertices in the subset (though it can contain more)



Steiner Tree Problem

Given a weighted graph and a subset of the vertices, find a tree of minimum weight that contains all vertices in the subset (though it can contain more)



Thank you!

Next time: Workshop 3: Graphs in the real world

Email me with questions: jolivier@stanford.edu